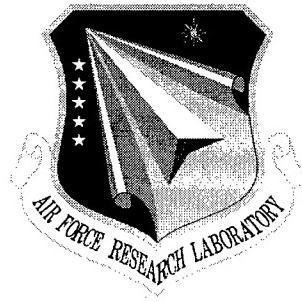


AFRL-IF-RS-TR-2000-118
Final Technical Report
August 2000



WINWIN EXTENSIONS FOR THE EVOLUTIONARY DESIGN OF COMPLEX SYSTEMS

University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D897

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

20001019 148

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

[DTIC QUALITY INSPECTED 4]

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-118 has been reviewed and is approved for publication.

APPROVED:



ROGER J. DZIEGIEL
Project Engineer

FOR THE DIRECTOR:



NORTHRUP FOWLER, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

WINWIN EXTENSIONS FOR THE EVOLUTIONARY DESIGN OF COMPLEX
SYSTEMS
Barry Boehm

Contractor: University of Southern California
Contract Number: F30602-96-2-0231
Effective Date of Contract: 18 July 1996
Contract Expiration Date: 30 June 2000
Short Title of Work: WinWin Extensions for the
Evolutionary Design of Complex
Systems
Period of Work Covered: Jul 96 – Jun 00

Principal Investigator: Barry Boehm
Phone: (213) 740-8163
AFRL Project Engineer: Roger Dziegel
Phone: (315) 330-3547

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This research was supported by the Defense Advanced Research
Projects Agency of the Department of Defense and was monitored
by Roger J. Dziegel, AFRL/IFTD, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
<p>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</p>			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED	
	AUGUST 2000	Final Jul 96 - Jun 00	
4. TITLE AND SUBTITLE	5. FUNDING NUMBERS		
WINWIN EXTENSIONS FOR THE EVOLUTIONARY DESIGN OF COMPLEX SYSTEMS	C - F30602-96-2-0231 PE - 62301E PR - D897 TA - 01 WU - 01		
6. AUTHOR(S)	8. PERFORMING ORGANIZATION REPORT NUMBER		
Barry Boehm	N/A		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		
University of Southern California Center for Software Engineering Los Angeles CA 90089-0781	Defense Advanced Research Projects Agency Air Force Research Laboratory/IFTD 3701 North Fairfax Drive 525 Brooks Road Arlington VA 22203-1714 Rome NY 13441-4505		
10. SPONSORING/MONITORING AGENCY REPORT NUMBER	11. SUPPLEMENTARY NOTES		
AFRL-IF-RS-TR-2000-118			Air Force Research Laboratory Projects Engineer: Roger J. Dziegier/IFTD/(315) 330-2185
12a. DISTRIBUTION AVAILABILITY STATEMENT		12b. DISTRIBUTION CODE	
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			
13. ABSTRACT (Maximum 200 words)			
<p>This research focused on formulation and development of process models and their support environments that will enable the DoD and its contractors to shift from traditional fixed-contract models of software and systems engineering in general to collaborative models of such processes. technology was developed to address DoD's increasing need for rapid, user-responsive concept definition, prototyping, development, and life-cycle evolution of complex software-intensive systems. The WinWin Spiral model uses Theory W (win-win) to develop software and system requirements, and architectural solutions, as win conditions negotiated among a project's stakeholders (user, customer, developer, maintainer, interfaicer). The WinWin negotiation tool is a Unix workstation-based groupware support system that allows stakeholders to enter win conditions, explore their interactions, and negotiate mutual agreements on the specifics of the new project being contracted. The model and support system also feature a central role for quantitative tradeoff analysis tools such as COnstructive COst Model (COCOMO) -- a tool which allows one to estimate the cost, effort, and schedule associated with a prospective software development project). For additional information on WinWin and EasyWinWin OnLine visit University of Southern California's Center for Software Engineering (USC/CSE) web page http://sunset.usc.edu/research/WINWIN/index.html. This project was funded by AFRL/IF and DARPA/ITO under Evolutionary Design of Complex Software (EDCS) Program.</p>			
14. SUBJECT TERMS		15. NUMBER OF PAGES	
WinWIN Spiral Model, Model-Based requirements Negotiation, Collaboration		108	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

1	Project Overview	1
1.1	WinWin Requirements Tool and Rationale Capture Extensions	1
1.2	WinWin Spiral and MBASE Product-Process Integration	2
1.3	Software Architecture Extensions and Integration with Requirements	2
1.3.1	Architecture Analysis Tools and Requirements Integration	2
1.3.2	Domain Architectures for Satellite Ground Systems	4
1.4	Experimental Application of Research Results	4
Appendix A: Easy WinWin: Lesson Learned from Four Generations of Groupware for Requirements Negotiation		7
Appendix B:	Spiral Development: Experience, Principles, and Refinements	21
Appendix C:	Life Cycle Connectors: Bridging Models Across the Life-Cycle	52
Appendix D:	Using the WinWin Spiral Model: A Case Study	62
List of Published Papers and Technical Reports		74
References		97

LIST OF TABLES

Table 1.	MBASE Project Experience at USC/Columbia	5
----------	--	---

1 Project Overview

1.1 WinWin Requirements Tool and Rationale Capture Extensions

On a previous DARPA contract, USC had developed two generations of an experimental WinWin tool to enable system stakeholders (e.g., users, customers, developers, maintainers) to negotiate a mutually satisfactory (win-win) set of system requirements [Boehm et al. 1995]. The tool captured the time history of these requirements negotiations, which provided a promising base for developing a Rationale Capture capability in support of the corresponding EDCS program goal.

The WinWin tool was re-architected and restructured to accomplish this goal and to effect a number of improvements identified in using the second-generation WinWin tool. These included:

- Development of a Rationale Graph linking the Win Conditions, Issues, Options, and Agreements involved in WinWin requirements negotiations, and providing a GUI capability to explore captured decision rationales by pointing and clicking on the Rationale Graph.
- Capturing rationale via Attachments, including architectural analyses, cost model tradeoffs, negotiation videoclips, scenario-based analyses, and executing software.
- Experimenting with Rationale Agents, which could reopen negotiations upon sensing the violation of an Agreement.
- Formulation and implementation of rigorous integrity conventions among negotiations, such as locking artifacts involved in votes, and disallowing out-of-sequence artifact creation.

The resulting WinWin tool was used successfully in over 100 real-client requirements negotiations at USC and Columbia U., and used experimentally by several DoD and contractor organizations, including USAF Space and Missile Systems Center, Warner Robins Air Logistics Center, AFRL/IFTD, Aerospace Corporation, TRW, Lockheed Martin, Northrop Grumman, Litton, SPC, and MCC. However, the tool was not used in continuing practice for several reasons:

- The rigorous integrity conventions made the tool too awkward to use in negotiation situations requiring flexibility.
- The server was Unix-based, and most organizations were using Windows-based servers.
- Although an applications program interface was developed, it was still hard to interoperate WinWin with other commercial groupware tools being used.
- The tool did not have a commercial support organization.

In the final year, with major contributions by Visiting Prof. Paul Gruenbacher, we entered a strategic partnership with GroupSystems.com, a leading commercial groupware company, to develop a commercial version of WinWin, called EasyWinWin, based on their groupware infrastructure. The resulting tool has been used enthusiastically by commercial companies, has been announced as a GroupSystems.com product, and is currently being introduced via a series of USC-GroupSystems training courses. Further details are provided by the paper in Section 2.1,

“EasyWinWin: Lessons Learned from Four Generations of Groupware for Requirements Negotiation,” currently submitted for publication in IEEE Software.

1.2 WinWin Spiral and MBASE Product-Process Integration

On the previous DARPA contract (F30602-94-C-0195), USC had developed an experimental WinWin Spiral process model which was used successfully to evolve the WinWin tool, but which did not fully integrate process and product definitions [Boehm and Bose 1994].

The WinWin Spiral model was extended into a full system development approach called Model-Based (System) Architecting and Software Engineering (MBASE) [Boehm and Port 1999b]. MBASE integrates not only a project’s product and process models, but also its property and success models, thus avoiding the model clashes found to underlay many large software project failures [Boehm and Port 1999a].

USC has developed roughly 200 pages of product definition guidelines for MBASE and an Electronic Process Guide based on the Software Engineering Institute (SEI)’s Electronic Process Guide (EPG) tool [Kellner 1999]. The MBASE guidelines have also been used successfully on over 100 real-client projects at USC and Columbia U. They have also been used by projects at TRW, Litton, and several commercial companies. They and the WinWin Spiral Model were identified as the spiral model version best suited for adoption for evolutionary acquisition of command and control systems by the Air Force General Officers’ Offsite on Spiral Development in February 1999. WinWin Spiral and MBASE principles were adopted by the 1 January 2000 Air Force Instruction 63-123, “Evolutionary Acquisition of C² Systems.”

In February 2000, USC and SEI co-sponsored a workshop at USC on “Spiral Development Experiences and Implementation Challenges,” attended by leading spiral practitioners in DoD and the aerospace and commercial sectors. The resulting conclusions and recommendations have become part of the current Deputy Undersecretary of Defense/Science and Technology (DUSD/S&T) initiative to provide technical infrastructure support for the software portions of DoD’s revised acquisition regulations DoDI 5000.1 and DoDI 5000.2. A follow on SEI-USC workshop, “Spiral Development in the DoD,” is being sponsored by the DUSD/S&T, Dr. Dolores Etter, in September 2000.

Further details, along with a set of defining invariants and variants for the spiral model, are provided by the paper, “Spiral Development: Experience, Principles, and Refinements,” presented as the keynote for the February 2000 workshop, and currently being prepared for publication as CMU/SEI-2000-TR-008.

1.3 Software Architecture Extensions and Integration with Requirements

1.3.1 Architecture Analysis Tools and Requirements Integration

Evolving system requirements into a viable software architecture is still mainly based on intuition with little available guidance. Requirements and the system architecture emerge in an iterative process involving multiple stakeholders with conflicting goals, needs, and objectives. An

important success-factor is thus to provide stakeholder-relevant views on the evolving requirements and architecture models. We found that a better understanding of requirements and architecture can significantly improve the quality and effort associated with software development. We have developed useful research results in the following domains:

- Architectural Support for Commercial-off-the-Shelf (COTS) Development: Composing software systems out of COTS components is becoming more popular but the risks of undesirable side effects these components might have onto other components are becoming increasingly severe. Although COTS components may run perfectly on their own, they may not work anymore when used together. Our work on the AAA architectural mismatch detection rules and tools assists in identifying potential problems early on to avoid software composition problems.
- Consistent Refinement: The problem of consistently engineering large, complex software systems of today is often a problem of transforming and validating requirements, architecture, design, and implementation models. Each software model is intended to highlight a *particular view* of a desired system. A combination of multiple models is needed to represent and understand the *entire system*. Ensuring that the various models used in development are consistent relative to each other thus becomes a critical concern. We have developed an approach that integrates and ensures the consistency across an architectural and a number of design models. The goal of our work is to combine the respective strengths of a powerful, specialized (architecture-based) modeling approach with a widely used, general (design-based) approach.
- MBASE Integration: Product models such as architecture and design have to be also seen in context of other types of models such as property models, success models, or process models. Our work on MBASE establishes criteria for evaluating development models in terms of key system stakeholders' concerns. The Model-Based (System) Architecting and Software Engineering (MBASE) approach enables consistent, concurrent definition of a system's architecture, requirements, operational concept, prototypes, and life cycle plans.
- Software Connectors: Software connectors describe the interactions among architectural components and support communication, coordination, conversion and facilitation needs of components. Connectors can be used to describe interactions among components in family architectures. Furthermore, improvements in many extra functional properties of a system can be attributed to semantically rich connector mechanisms such as events, distributors and arbitrators. Since connectors can be applied across problem domains, they have a high potential for reusability. Connectors also significantly affect global system properties such as availability, throughput, security and scalability.
- Model Connectors: Numerous notations, methodologies, and tools exist to support software system modeling. While individual models may clarify certain system aspects, the large number and heterogeneity of models may ultimately hamper the ability of stakeholders to communicate about a system. The major reason for this is the discontinuity of information across different models. In our work we found an approach for dealing with that discontinuity. As such we built a set of extensible "connectors" to bridge models, both within

and across the activities in the software development lifecycle. While the details of these connectors are dependent upon the source and destination models, they share a number of underlying characteristics. We have illustrated our approach by applying it to a large-scale system we are currently designing and implementing in collaboration with a third-party organization.

1.3.2 Domain Architectures for Satellite Ground Systems

We organized a series of Ground System Architecture Workshops (GSAW), in 1997, 1998, and 1999. Presentations at the workshops included EDCS-funded projects highlighting relevant research, Government policy and plan briefings and the contractors describing the state of the practice as well as their research directions. All presentations at each GSAW have been posted on the USC website and published in the Proceedings of GSAW 97, GSAW 98, and GSAW 99.

Before the first GSAW in 1997, there had been no regular forum for the SGS community to exchange information, ideas, and issues relating to architectures for spacecraft ground systems. By the time of GSAW 99, a regular following had already been built up. The EDCS-funded series of workshops, GSAW 97, GSAW 98, and GSAW 99, has been extremely successful in creating and establishing a community of satellite ground system practitioners committed to investigating architecture technology and impact in the DoD and related Government and commercial systems.

GSAW2000 was held the year following the last EDCS-funded GSAW, and was well attended by both repeat participants, and new attendees. Like its predecessors, it was judged to be extremely valuable, both for the content of the briefings and breakout groups, and for the contacts with others in the coalescing spacecraft ground systems community. The workshops will be continued beyond the completion of the EDCS contract, and will continue to both provide a forum for technology transfer for DARPA and other researchers and also inform spacecraft ground system practitioners of issues and advances in related technologies.

We developed several architectural models using the Stanford Architecture Definition Language, Rapide. The architecture and the components that we modeled were drawn from an Aerospace Technical Operating Report describing a proposed reference architecture for the Standard Satellite Control Segment. This report was distributed within the DARPA community and published on the USC website. The models served as the basis for several studies, which we reported on in the Ground System Architecture Workshops. The evaluation criteria identified in these studies and in other work were collected and published in a report on Evaluation Criteria for Satellite Ground System Architectures, Aerospace Technical Report ATR-99(7470)-1.

1.4 Experimental Application of Research Results

During each year of the project, the evolving WinWin tools, MBASE guidelines, and selected architecture capabilities were used and evaluated on sets of real-client digital library applications at USC and related applications at Columbia U. The capabilities were strengthened and extended based on the year's experience, and applied in the following year.

Table 1. MBASE Project Experience at USC/Columbia

Metrics	USC 1996- 1997	USC 1997- 1998	USC 1998- 1999	columbia U-grad. 1999	columbia Grad. 99	USC 1999- 2000
Fall Semester: LCA Package						
Teams	15	16	20	20	13	21
Students	86	80	102	107	59	102
Applications	12	15	17	10	10	20
Teams failing LCO review	4	4	1	10	6	0
Teams failing LCA review	0	0	0	0	1	0
Pages, LCO package	160	103	114	124	116	TBD
Pages, LCA package	230	154	167	142	142	TBD
Client Evaluation (1-5, 5 best)	4.46	4.67	4.74	-	-	4.55
Spring Semester: IOC Package						
Teams	6	5	6	Remained the same since projects were only one semester long		8
Students	28	23	28			39
Applications	8	5	6			8
Teams failing IOC acceptance review	0	0	0	0	0	0
Applications satisfying clients (*teams)	5	5	6	20*	12*	8
Applications not overtaken by events	6	4	4	10	9	TBD
Applications continued	3	3	4	-	-	TBD
Applications used	1	3	3	10	5	TBD
Client evaluation (1-5, 5 best)	-	4.15	4.3	4.44	4.21	TBD

* LCO: Life Cycle Objectives; LCA: Life Cycle Architectures; IOC: Initial Operational Capabilities

Table 1 summarizes the experience on these projects. It shows a strong general progression toward successful passing of reviews, client satisfaction, and applications transitioning to successful use.

More detailed analyses of particular experiences, such as the WinWin negotiations, have been performed and documented in such papers as [Boehm and Egyed 1998] and [Egyed and Boehm 1999]. Further details on the digital library applications are provided by the paper in Section 2.4, “Using the WinWin Spiral Model: A Case Study,” published in IEEE Computer.

Research Area Summary Papers

Appendix A:	Easy WinWin: Lesson Learned from Four Generations of Groupware for Requirements Negotiation	7
Appendix B:	Spiral Development: Experience, Principles, and Refinements	21
Appendix C:	Life Cycle Connectors: Bridging Models Across the Life-Cycle	52
Appendix D:	Using the WinWin Spiral Model: A Case Study	62

EasyWinWin: Lessons Learned from Four Generations of Groupware for Requirements Negotiation

Barry Boehm

Paul Gruenbacher

Robert O. Briggs

*Computer Science Department
University of Southern California
941 W. 37th Place, Los Angeles, CA 90089-0781
{boehm, gruenbac}@sunset.usc.edu*

*GROUPSYSTEMS.COM
1430 E. Fort Lowell Rd. #301,
Tucson, AZ 85719,
bbriggs@groupsystems.com*

1 Introduction and Motivation

There is no complete and well-defined set of requirements waiting to be discovered in system development. Requirements emerge in a highly collaborative process that involves users, customers, managers, domain experts, and developers. These stakeholders contribute incomplete, vague, and inconsistent statements and ideas about their objectives, assumptions, and expectations. Requirements negotiation is therefore essential to achieve mutually satisfactory agreements and to elaborate complete, correct, and clear specifications.

Collaborative technology supporting this process has to address the heterogeneity of stakeholders in particular: Groupware systems are among the hardest of systems to get right. The rapidly moving technology of distributed interactive systems is a major challenge. However, even bigger is the challenge of creating a system that works well with people of different backgrounds, in different places, and often at different times.

Here we present the major lessons we have learned in developing four generations of a distributed groupware system called WinWin. We also summarize the degree to which the current system, EasyWinWin, satisfies the original and evolving objectives for such a system, and demonstrate how it enables and facilitates active participation and stakeholder collaboration.

What is the WinWin approach?

The WinWin approach evolved more or less independently as an interpersonal relations [16], success management [7], and project management [1] approach. A reasonable common definition is:

The WinWin approach is a set of principles, practices, and tools, which enable a set of interdependent stakeholders to work out a mutually satisfactory (win-win) set of shared commitments.

The interdependent stakeholders can be either people or organizations. Their shared commitments can be about information system requirements (the primary focus of the WinWin groupware system), but can cover any continuing relationships in work and life. "Mutually satisfactory" generally means that people do not get everything they want, but that they can be reasonably assured of getting what was agreed to. "Shared commitments" are not just good intentions, but carefully defined conditions. If someone has a conditional commitment, the condition needs to be made explicit, and understood as part of the agreement by all stakeholders.

Why does WinWin work?

The alternatives don't work

In a requirements negotiation, nobody wants a lose-lose outcome. Win-lose may sound attractive to the party most likely to win, but it usually turns into lose-lose. Table 1 shows three classic win-lose patterns

among the three primary system stakeholders – developers, customers, and users – in which the loser’s outcome usually makes the two “winners” into losers as well [2].

Table 1: Frequent Software Development Win-Lose Patterns (which usually turn into lose-lose situations)

Proposed Solution	“Winner”	Loser
Quick, Cheap, Sloppy Product	Developer & Customer	User
Lots of “bells and whistles”	Developer & User	Customer
Driving too hard a bargain	Customer & User	Developer

Building a quick and sloppy product may be a low-cost, near-term win for the software developer and customer, but it will be a lose for the user (and the maintainer). Adding lots of marginally useful “bells and whistles” to a software product on a cost-plus contract may be a win for the developer and users, but is a lose for the customer. And “best and final offer” bidding wars imposed on competing developers by customers and users generally lead to low-ball winning bids which place the selected developer in a losing position.

Actually, nobody wins in the above situations. Quick and sloppy products destroy a developer’s reputation and have to be redone, inevitably at a higher cost to the customer. The “bells and whistles” either disappear or (worse) crowd out more essential product capabilities as the customer’s budgets are exhausted. Inadequate low-ball bids translate into inadequate products, which again incur increased customer costs and user delivery delays to reach adequacy.

WinWin builds trust and manages expectations

If you consistently find other stakeholders asking about your needs and acting to support them, you will end up trusting them more. In addition, if you consistently find them balancing your needs with other stakeholders’ needs, you will have more realistic expectations about getting everything you might want.

WinWin helps stakeholders adapt to changes in the environment that effect requirements

Our traditional, adversarial, lawyer-oriented contracting mechanisms are no match for our current world of increasing rapid change in technology, mergers, reorganizations, and personnel turnover. Instead of rigorous requirements in ironbound contracts, doing business in Internet time requires stakeholders with a shared vision and the flexibility to quickly renegotiate a new solution once unforeseen problems or opportunities arise. A WinWin approach builds a shared vision among stakeholders, and provides the flexibility to adapt to change.

WinWin helps to build institutional memory

The *why* behind the *what*, i.e., the decisions that led to a work result often vanish. By capturing stakeholder negotiations WinWin supports long-term availability of the decision rationale and helps to build institutional memory.

How does the WinWin System work?

The particular WinWin system we have evolved includes a negotiation model for converging to a WinWin agreement, and a WinWin equilibrium condition to test whether the negotiation process has converged: The negotiation model guides success-critical stakeholders in elaborating mutually satisfactory agreements. Stakeholders express their *win conditions*. If everyone concurs, the win conditions become *agreements*. When stakeholders do not concur, they identify their conflicted win conditions and register their conflicts as *issues*. In this case, stakeholders invent *options* for mutual gain and explore the option trade-offs. Options are iterated and turned into agreements when all stakeholders concur. Additionally, a

taxonomy is used to organize WinWin artifacts. Important *terms* of the domain are captured in a glossary. The stakeholders are in a WinWin equilibrium condition when all of their win conditions are covered by agreements, and there are no outstanding issues.

This negotiation model provided the basis of all four implementations of WinWin groupware systems. We will present these implementations and will show how the current system, EasyWinWin, addresses the shortcomings and lessons learned in the development of the previous systems. Our major lesson learned is that collaborative technology for requirements engineering has to be based on a sound methodology and on proven collaboration and facilitation techniques that emphasize group dynamics. We have been addressing these challenges in the EasyWinWin project: We will introduce the concept of thinkLets and show how thinkLets are used in the EasyWinWin methodology. We will also present practical experiences gained with EasyWinWin.

2 Four Generations of WinWin Groupware and Lessons Learned

The original motivation for a WinWin groupware system was the first author's frustration in using a manual WinWin approach to manage large projects at DARPA. For example, WinWin management of the \$100 million DARPA STARS program was done primarily via monthly meetings of many STARS stakeholders: 3 prime contractors and their 3 commercial counterparts; 3 user representatives from the Army, Navy, and Air Force; DARPA customers, contract managers, and several research and support contractors. Each meeting would end up with a WinWin agreement that felt like three steps forward. However, by the next meeting, we would take two steps back, as the distributed stakeholders independently "reinterpreted" the agreements.

As a result, it took six months to achieve a shared vision documented by the prime contractor's success plans. Our analysis at the time indicated that an "anytime, anywhere" WinWin groupware support system could have reduced this to 1-2 months.

DARPA's maturing Internet technology looked like a good technology base for developing such a system. USC's Center for Software Engineering, initially with its Affiliates' support, later with DARPA and Air Force Research Labs' support and a grant by the Austrian Science Fund (second author), developed a series of four WinWin groupware implementations. These reflect increasing understanding of what was needed for successful WinWin groupware operations and technology support.

2.1 Generation 1: Initial Prototype

The first WinWin groupware implementation was a prototype developed in concert with Perceptrronics' CACE-PM® support system for concurrent engineering of multi-chip modules. CASE-PM® enabled us to develop a useful rapid prototype, which was useful enough for demonstrations and for an initial experiment. This involved the system's developers role-playing as future system developers, customers, and users negotiating the requirements for a more robust version of WinWin. The main lessons learned from this experiment were [2]:

- The WinWin approach helps bridge a previous gap in using the Spiral process model: how to determine the next round of objectives, alternatives, and constraints. This led to the WinWin Spiral Model extensions [2, 3], now used by several organizations.
- Software requirements negotiation required considerably more database and relationship management than was needed for multi-chip modules. This led to a much more thorough definition of WinWin artifacts and relationships, including the basic negotiation model discussed above.
- In exploring experimental use of the system by Affiliates, we found that the high cost of the CACE-PM® license was a major disincentive (USC had a free academic license). This led us to focus on a small-footprint USC-built system.
- Performing the WinWin negotiation gave us a strong, shared vision for the next version of the system, validating its utility as a groupware capability.

2.2 Generation 2: Strong Vision, Not-So-Strong Architecture

The second-generation WinWin system used a Sun-UNIX client-server architecture, X/Motif GUI support, and its own database server. It was used experimentally by some friendly industry users. Its main value was to identify a number of inconsistencies in the negotiation model and the artifacts, among the artifacts, and between GUI and the database server. We had underestimated how much detailed software engineering was needed to get from a shared groupware vision to a groupware support system.

2.3 Generation 3: Muscle-Bound Architecture

The third-generation WinWin system had a formally analyzed negotiation model, uniform artifact look and feel, carefully defined GUI-database interfaces, and rigorous enforcement of the negotiation model. For example, one could not define Issues without Win Conditions as referents, or Options without Issues as referents. When an Agreement was put to a vote, all of its associated Win Conditions, Issues, and Options were locked to preserve integrity of the voting process. The 3G WinWin also had a number of amenities for voting, for attaching associated documents or analysis-tool runs, and for big-picture negotiation visualization and navigation.

The 3G WinWin was sufficiently robust to have supported four years' work of 15-20 project negotiations per year [3, 8]. These involved USC librarians and student teams negotiating the requirements for operational USC digital library systems, which the student teams then built and transitioned to library use. In the first year, we learned not to do the WinWin negotiations ahead of the prototype, as we rediscovered the IKIWISI (I'll know it when I see it) syndrome. Once the librarians saw the prototype, they wanted to re-do all the negotiations. Still, we verified across over 100 requirements negotiations that 3G WinWin was able to support rapid definition and development of unprecedented applications.

3G WinWin was also used experimentally by several industry and government organizations. However, this use did not lead to the system's crossing the chasm into mainstream use. The main reasons cited by the mainstream users were:

- 3G WinWin's integrity rules were too rigorous. For example, to fix a typo in an artifact that was locked for voting, users had to make all the locked artifacts inactive and copy their contents into a new set of artifacts.
- The homebrew database server was very fragile and prone to lose people's work in crashes.
- 3G WinWin did not interoperate well with other groupware systems, even after we built an applications program interface for it. One industry project successfully built and applied its own WinWin overlay on top of the groupware system it was using, but did not try to develop a more general capability.

2.4 Generation 4: EasyWinWin and thinkLets

These experiences turned USC more toward developing a version of WinWin based on a commercial groupware infrastructure developed by GroupSystems.com in cooperation with the University of Arizona [13]. Our current collaboration between USC and GroupSystems.com has led to a fourth-generation system, EasyWinWin [11, 5], which is proving successful on mainstream industry applications. It is described in the next sections.

3 thinkLets™ for stakeholder collaboration

Each of our first three generations of WinWin groupware was increasingly strict about enforcing modeling conventions at the expense of group dynamics. The software got in the way of the human interactions that are a critical part of negotiation. In the EasyWinWin project we relaxed modeling constraints and focused instead on moving people through a comfortable process that focused their attention on just the right subtask at just the right time. A decade of research revealed that the use of collaborative technology could produce stunning gains in productivity (see [10] for a thorough review of Group Support Systems Research). However, the technology on its own was not sufficient to assure predictable, repeatable

success. Different teams using the same technology for the same task would develop dramatically different group dynamics and would vary dramatically in their productivity based solely on differences in the facilitation script – the directions given to the group at the beginning of a collaborative activity [15]. From this experience, one of us (Briggs) developed the concept of thinkLets. A thinkLet encapsulates all the intellectual capital required to create one predictable, repeatable pattern of group interaction.

A thinkLet has three components: A collaborative reasoning tool, a configuration, and a carefully crafted and tested facilitation script. The first component of a thinkLet, a GroupSystems tool, seems simple on the surface:

Electronic Brainstorming: Team participants contribute comments to electronic pages and repeatedly exchange those pages to stimulate additional thoughts and comments (see Figure 2).

Categorizer: People drag-and-drop ideas into *buckets* and contribute to discussion pages attached to each idea in order to organize their thoughts and elaborate on their concepts (see Figure 3).

Topic Commenter: This tool provides a stack of electronic cards, each with a different topic heading. It encourages team members to explore a tightly bounded set of concepts in depth and detail.

Group Outliner: The Group Outliner allows a team to build a tree structure of ideas and concepts (see Figure 1 and Figure 4).

Vote: Team members vote a list of items using one of 10 different methods, or they create a custom voting method on the fly.

Alternative Analysis: Participants assess a list or tree of items against a set of criteria.

Survey: A team collects responses to many kinds of questions and analyzes the results.

Although the tools appear simple to the users, their design derives from cognitive and social research findings and from extensive experience in the field with teams trying to accomplish meaningful work.

The second component of a thinkLet is the configuration of the collaborative reasoning tool. Each tool in GroupSystems suite has 15 to 25 configurable features that may be varied independently, yielding more than 9 million possible configurations. Each configuration has some subtle, yet powerful impact on group dynamics. When designing a thinkLet, a methodologist can select an appropriate configuration to create just the right dynamics for the task.

The third component of a thinkLet is its facilitation script. Any given tool and configuration can be used in many ways by a team. The script explains how team members should respond to the tool before them. Different scripts will produce very different results. For example, below is an excerpt from the script for the GroupSystems FreeBrainstorming thinkLet, which encourages the participants to:

"...argue with one another, expand on one another's ideas, or be inspired to contribute completely new ideas; contribute as many new ideas as you can in a short time."

Given that guidance, and the appropriately configured tool, a group tends to push quickly to the boundaries of their problem space and discover new ideas that are outside familiar patterns of thinking.

The *Point-Counterpoint* thinkLet produces completely different dynamics with the same tool and configuration. This thinkLet causes a group to find the middle ground when they have become polarized and hit an impasse. The script excerpt below encourages the group to:

"... put in your strongest single argument in favor of your position. Now trade pages. You should see somebody else's argument on your screen. Whether you agree with the argument or not, type in the strongest counter argument you can against the first argument on the page. Now trade pages. You should see a strong argument and a mutually exclusive counter argument. Now write an argument that bridges those seemingly irreconcilable positions."

All three components of GroupSystems thinkLets – tool, configuration, and technique – emerge from years of field experience and from more than a decade of cognitive, organizational, and social research

into the foundations of successful collaboration. Because each thinkLet produces a predictable, repeatable pattern of interaction, a methodologist can assemble the thinkLets into a sequence of steps to guide a group through a cognitively challenging task like requirements negotiation.

There are thinkLets for six different categories of group dynamics: Diverge, Converge, Organize, Evaluate, Elaborate, and Agree. Each step in a group process requires some different pattern of interaction. For example, a group working to solve a problem might *diverge* from customary thinking patterns in search of new solutions. They may *evaluate* their options, *converge* on a workable solution, and *elaborate* their plan. With each category, there are thinkLets to cause subtle yet powerful variations in the overall pattern. Consider, these three examples of convergence thinkLets:

FastFocus causes a group to quickly extract a small set of the most important issues from a vast, unstructured brainstorming transcript.

GoldMiner helps a team identify key issues while keeping the issues in the context from which they emerged.

BroomWagon guides a group to take a set of ideas through multiple rounds of elimination until only the most desirable ideas remain.

There are also thinkLets that cut across several categories. The Could-Be-Should-Be thinkLet, for example, takes a group through a series of diverge/converge cycles. Each cycle produces a new layer of detail.

We used and refined nine different thinkLets to implement a repeatable, reliable WinWin methodology. The implementation is described in more detail in the next section.

4 The EasyWinWin methodology

This section describes the nine stages in the EasyWinWin methodology:

- Elaborating the domain taxonomy
- Brainstorming stakeholder interests
- Converging on win conditions
- Capturing domain language
- Prioritizing win conditions
- Elaborating conflicts and constraints
- Elaborating options
- Negotiating agreements
- Mapping negotiation results to the taxonomy

Elaborating the domain taxonomy

Stakeholders tend to enter software development projects with vague and limited understandings of what can and should be done. The first step in the EasyWinWin methodology is therefore to broaden their perspectives by asking them to elaborate, refine, and customize the *domain taxonomy*. The taxonomy serves as both an organizing framework and a stakeholder checklist for WinWin negotiations. It becomes a useful way to organize the hundreds of win conditions that emerge later in the project, and the resulting requirements specification.

When used with the USC Model-Based (System) Architecting and Software Engineering (MBASE) approach, the domain taxonomy corresponds to the Table of Contents of the requirements specification [4]. MBASE defines the following sections: *Application capabilities* (system features and services), *Interfaces* (to the user as well as other software and hardware systems), *System Properties* ('non-functional' requirements), *Project & process constraints* (cost, schedule, development tools, support), and *Evolution* (most likely directions of requirements change and growth).

Stakeholders suggest modifications to the taxonomy to reflect their own interests and culture. EasyWinWin uses the *Could-be/Should-be* thinkLet: The taxonomy appears on the stakeholder's screens as a shared outline (Figure 1). A double-click of any outline node opens a shared comment window for that

node. During the ‘could be’-step, stakeholders add comments about what might be missing in the taxonomy, or what should be removed from the taxonomy. During the ‘should be’-step a moderator reviews these comments together with the group and modifies the outline itself.

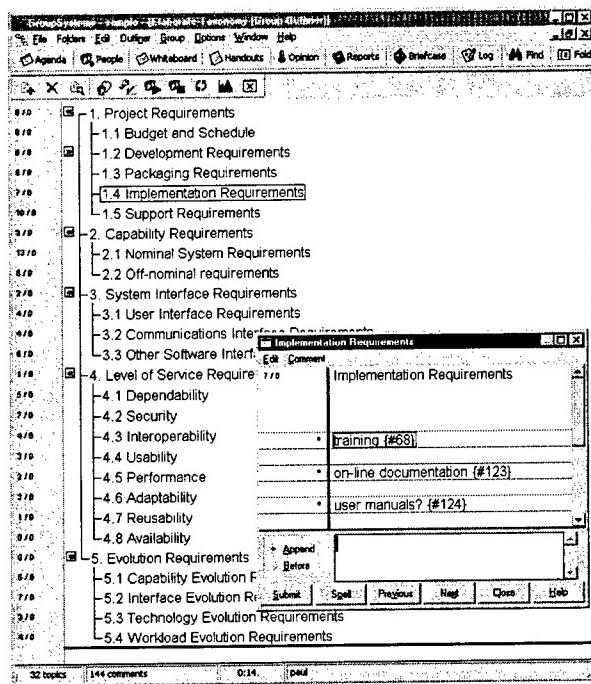


Figure 1: Elaborate domain taxonomy

Brainstorming stakeholder interests

In this activity, stakeholders are encouraged to exhaustively explore their vested interests in the project. Using the *FreeBrainstorming* thinkLet in the GroupSystems Electronic Brainstorming Tool, stakeholders make anonymous contributions to a series of electronic pages that swap randomly among their screens (Figure 2).

Figures 2, 3, and 4 show sample EasyWinWin sessions: The USC bookstore together with a consortium of other university bookstores is developing a web-based portal for students. EasyWinWin was used to negotiate requirements and to develop consensus about the elements and features of the portal.

Rapid, simultaneous, and anonymous interaction allows sharing different opinions and perspectives, and exploiting the creative potential and experiences of the team. With this thinkLet, the group can hear from all its members in less time than it would normally take to hear from just one or two people. Typical sessions last between 30 and 60 minutes. In our experiences, the teams identified between 100 and 300 ideas during this phase. As contributions are anonymous at this point, people reveal interests that they might be reluctant to discuss in an identified brainstorming session. The resulting collection of stakeholder statements and ideas provides a starting point for building win conditions and defining important terms of the domain.

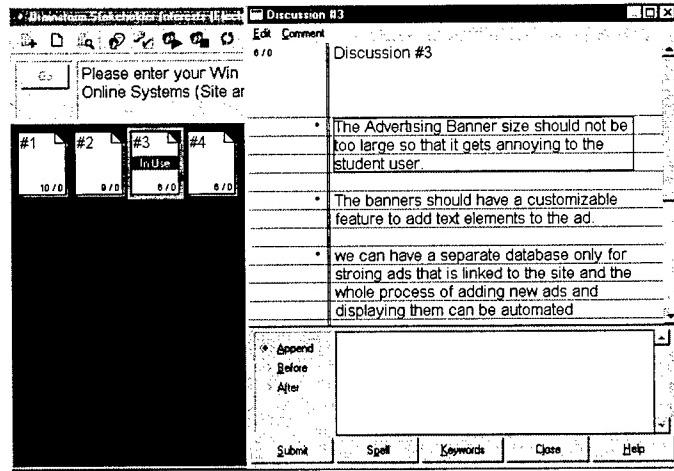


Figure 2: Brainstorm stakeholder interests

Converging on win conditions

Brainstorming comments from stakeholders are usually unstructured, redundant, ambiguous, and vague. The goal of this activity is to jointly craft a non-redundant list of clearly stated, unambiguous win conditions by considering all ideas contributed in the brainstorming session. Stakeholders also organize these win conditions into buckets that represent the top-level domain taxonomy elements.

The team uses the *FastFocus* thinkLet for converging on the list of win conditions. Each stakeholder views a different electronic page containing twenty-to-thirty comments contributed by the group. The moderator prompts each stakeholder in turn to orally frame a succinct statement of a win condition from the list of comments. Typically, several brainstorming statements are merged into one win condition. However, if a statement covers several aspects (e.g., it describes a system service and a system property), it may result in several win conditions. The moderator types the win condition onto a publicly-viewable list (see Figure 3), and the group discusses the statement to be sure all stakeholders understand its meaning. The group is explicitly prohibited from arguing with or endorsing win conditions at this time. Once each participant has had a chance to contribute a win condition, the system randomly swaps pages among the participants. The moderator repeats the pattern, asking stakeholders to frame win conditions from the screens before them. The process continues until no participant can find any new win conditions to add to the list. The number of win conditions is typically about one third of the number of brainstorming ideas.

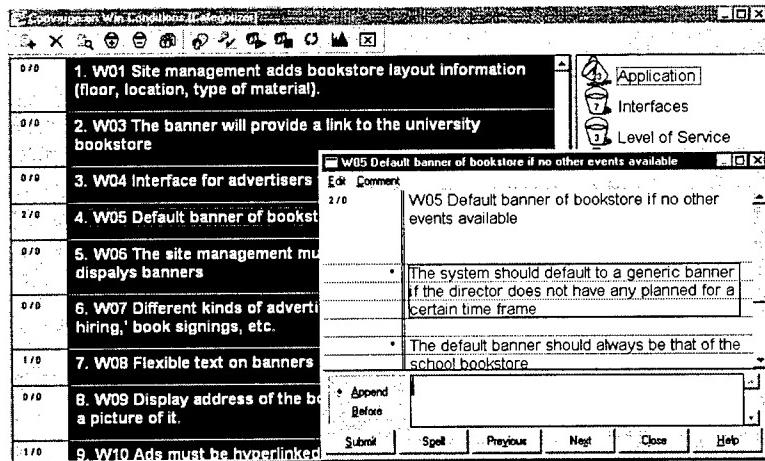


Figure 3: Converge on win conditions

Capturing domain language

As participants brainstorm, they use words that have special meanings within the context of a project or a domain. During the convergence step, the moderator adds important terms to a shared list in the electronic brainstorming tool. The groupware then automatically organizes brainstorming comments containing these keywords (or synonyms of the keywords) on different discussion sheets. Each sheet shows how a certain term is used in different statements and ideas. Stakeholders use this information to create and jointly review definitions for these terms.

Prioritizing win conditions

Voting in EasyWinWin serves two major purposes: (1) Determining priorities of win conditions and (2) revealing conflicts and different perceptions. The team evaluates the win conditions using the *MultiCriteria* thinkLet. Stakeholders rate each win conditions on a scale from 1 to 10 for each of two criteria: *Business importance* shows the relevance of a win condition to project success; *ease of realization* indicates perceived technical or economic constraints of implementing a win condition. In the voting process, developers focus on technical issues, while clients and users rate the business relevance. The system automatically tabulates the results and displays the win conditions in one of four categories:

“*Low hanging fruits*”: Win conditions with a high business importance and low expected difficulties.

“*Important with hurdles*”: High-priority win conditions that are difficult to implement.

“*Maybe later*”: Low-priority win conditions that may be considered later because they will be easy to execute.

“*Forget them*”: Unimportant win conditions that are difficult to achieve.

Elaborating Conflicts and Constraints

Stakeholders examine the results of the prioritization poll on their screens. The *Crowbar* thinkLet is used to analyze patterns of agreement and disagreement. The system displays items with high consensus in green, and items with low consensus in red. The moderator shows a graph of voting patterns for any red item, and asks the group to speculate why the disagreement exists as follows:

“Without telling me how you voted, what reasons might exist for rating this win condition as very important, and what reasons might exist for rating it as unimportant?”

As stakeholders try to explain why opinions may have differed, they discover and correct ambiguity in their win conditions. They surface unspoken constraints, unrealized assumptions, and unshared information. A scribe records these items as comments, Issues (and sometimes Options) for the win condition under discussion.

Elaborating Issues

Conflict identification and resolution in the WinWin negotiation model is based on Issues, Options, and Agreements. Stakeholders identify these artifacts in several iterations and organize them in the GroupOutliner (Figure 4). The moderator encourages the team members to read each win condition. If the win condition raises any issue for the stakeholder, the stakeholder may type the issue as a sub-heading to the win condition. At this point, the participants may not argue about the issues they raise. They may only record their issues. The deliverable is a tree with win conditions as main headings and issues as subheadings.

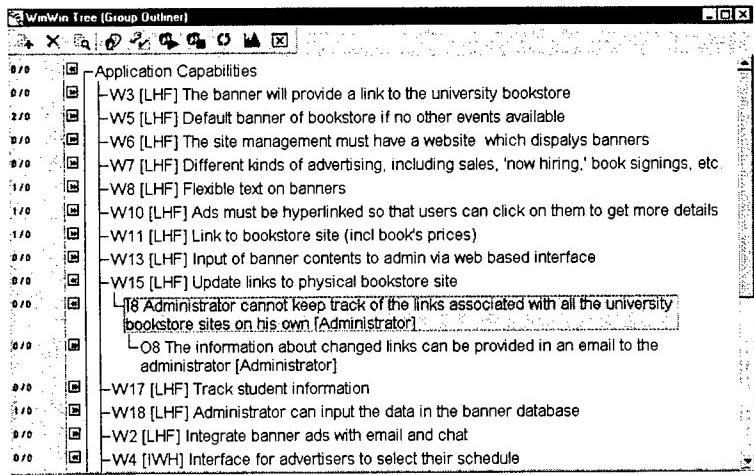


Figure 4: WinWin negotiation tree

Elaborating Options

The moderator encourages the stakeholders to read each issue. If the stakeholders can think of any options for addressing an issue, they may enter the option as a subheading to the issue. Participants are still discouraged from arguing about issues or options at this phase. The deliverable is a tree with win conditions, some of which will have issues as subheadings, some of which will have options as sub-subheadings. This deliverable is called the WinWin tree.

Negotiating Agreements

Win conditions for which no issues have been raised are usually automatically declared agreements. The group must negotiate a resolution to every issue in the WinWin tree. Often someone in the team has already proposed an option that can be used to resolve the issue. These options typically become agreements. Other issues cannot be resolved easily. These must be negotiated by the old-fashioned way – horse-trading, persuasion, lobbying, inventing new options for mutual gain, or give-and-take. However, because the lead-in process tends to be exhaustive, the discussion can be focused, and the available trade-offs are explicit. Because stakeholders are likely to have several automatic wins early in the process, they tend to be willing to make trade-offs with one another. The group continues to work until the WinWin equilibrium is achieved and all issues and win conditions are covered by agreements.

Mapping negotiation results to taxonomy

This technique helps to organize the negotiation results and to test if all aspects in the taxonomy have been sufficiently covered in the process. Stakeholders associate all WinWin artifacts with the domain taxonomy elements using the *PopcornSort* thinkLet: The moderator copies all the WinWin artifacts to a new list. Next to this list is a set of electronic buckets, each labeled with a heading from the domain taxonomy. Any team member may click-and-drag any artifact into any bucket. The process moves quickly; a team can sort a list of 200-300 artifacts in less than five minutes. Using the *BucketWalk* thinkLet, the team then reflects the contents of each bucket (taxonomy element) to decide whether any artifact should be moved to a different bucket, and whether any item needs to appear in more than one bucket. The team also verifies if all taxonomy elements have been thoroughly explored.

Figure 5 depicts the EasyWinWin activities and the artifacts connecting the activities. Solid lines show artifacts that are passed between activities for further refinement. Dashed lines indicate artifacts that are viewed in other activities without modification.

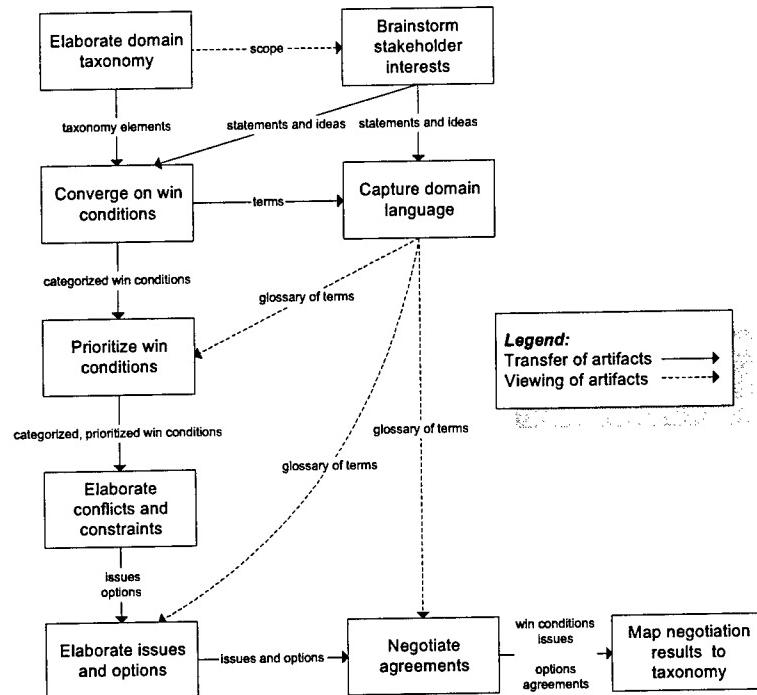


Figure 5: EasyWinWin activities

5 EasyWinWin experiences

EasyWinWin was validated in real-client projects. We applied the methodology in various domains (e.g., digital libraries, e-Marketplace, collaboration technology) and thoroughly explored and refined different thinkLets with the goal of streamlining the negotiation protocols and the overall order and design of process steps. EasyWinWin has been used in various contexts of requirements definition: Examples include the development of a shared vision, requirements definition for custom development projects, COTS acquisition and integration, transition planning, as well as COTS product enhancement and release planning. Table 2 summarizes some projects and their characteristics.

Table 2: Project Characteristics

<i>Project</i>	<i>Domain (Customer)</i>	<i>System characteristics</i>	<i>Requirements context</i>
Multimedia databases for instruction and research	Digital Library (Center for Scholarly Technology)	Web-based information system	Custom development requirements definition
Business/reference Q&A's question-matching-answer strategies for customer support	Digital Library (USC Library)	Web-based information system	Development of shared vision
Virtual calendar	Web Portal (Bookstore Consortium)	Web-based information system	COTS acquisition & integration
Email & student-to-student chat capabilities	Web Portal (Bookstore Consortium)	Web-based collaboration capability	COTS acquisition & integration
Ad management	Advertising (Bookstore Consortium)	Web-based information system	Custom development requirements definition
Infrastructure for media buying and selling	Media (Mediaconnect)	Web-based infrastructure for e-Marketplace	Web-Portal Definition
GroupSystems COTS Product Planning	Groupware (GroupSystems.com)	Commercial off-the-shelf product	COTS Product Enhancement

EasyWinWin experiences and benefits can be summarized as follows:

Repeatable process

The EasyWinWin process guide [9] describes how to use proven thinkLets in a requirements negotiation process. Software engineers experience less variance in the quality of their deliverables and the success of their engagements. Lower skilled or less experienced practitioners can accomplish more than would be possible if they had to do straight stand-up facilitation.

Improved stakeholder involvement

EasyWinWin allows direct participation of more people and elicits more from everybody involved. This leads to better buy-in as more interests can be accommodated earlier in the process. It also helps to develop broader and deeper deliverables: We found that EasyWinWin results surpass G3 WinWin results in terms of number of artifacts that are collected in a negotiation [8]. The higher number of issues identified and resolved helps to reduce risks early in a project and do not derail it later.

Velocity

We also found that EasyWinWin requires fewer stakeholder hours than previous WinWin generations. Rapid stakeholder interaction reduces costs needed to execute the engagement. As fewer hours are required, higher-ups are more willing to be involved. People with the power to cut deals tend to be present to cut the deals. This leads to improved decisions and a lower chance of expensive rework.

Institutional memory

Electronic transcripts providing a record of previously ephemeral content help to preserve the full decision rationale and avoid the drudgery of compiling meeting minutes after the engagement. Developers and customers also benefit from instant availability of results. Customers see success much earlier in the engagement. Decisions are more auditable and result in more detailed, accurate, and complete deliverables.

Anonymity

Anonymous submission of win conditions and revelation of conflicts and constraints fosters candor. People with power differentials can lay it on the line without threat to job, status, relationships, and political position. Increased openness also helps to get to the root issues in a hurry. People up and down the hierarchy can find out what is really going on thus avoiding the Abilene Paradox (people agree to an unattractive option because they erroneously believe it will make the option-proposer happy) [12].

Mode of collaboration

Beyond same-time/same-place stakeholder interaction, we successfully included remote participants into EasyWinWin workshops by utilizing the web-based capabilities of GroupSystems and additional use of audio links. We are currently elaborating recommendations for geographically distributed teams intending to adopt EasyWinWin activities in different time/different place setting.

6 Conclusions

We would single out five primary groupware lessons learned from the four generations of experience with WinWin.

Build on facilitation and collaboration techniques

The first three generations of WinWin environments emphasized modeling constraints over group dynamics and collaboration support. ThinkLets adopted in EasyWinWin provide a mechanism to define repeatable patterns of group interaction.

Use the system to plan its own future

It provides both a good test of the current groupware system and a good way of achieving a shared vision of its future directions. Both USC's experience with 1G WinWin and GroupSystems.com's experience with EasyWinWin substantiates this.

Make sure your stakeholders are representative, empowered, knowledgeable, collaborative, and committed

These characteristics came out of our critical success factor analysis of which digital library projects have succeeded or failed in being actually used. It often involves pre-screening of stakeholder negotiations, and shared-knowledge-building activities such as concurrent prototyping.

A similar conclusion holds with respect to the representativeness of the user base for which you are building the groupware system. Once we had an annual set of USC projects to support with 3G WinWin, we over-focused on USC users rather than our primary target of mainstream industry users.

When developing groupware, perseverance pays off

Do not overreact to initial negative experiences. Groupware systems must be carefully balanced to accommodate the many needs of the many stakeholders. In the design of 2G and 3G WinWin, we reacted to the 1G WinWin experience with its high-priced commercial infrastructure by building homebrew infrastructure. The 4G EasyWinWin overlay above GroupSystem's infrastructure has been much more successful.

In the design of 3G WinWin, we also overreacted to some instances of artifact misuse by creating a system whose rules were so rigorous that they turned off most users. 3G WinWin improved over 2G WinWin with its well-defined architectural interfaces, but lost out because of its inflexibility for mainstream stakeholder groups.

Relative to the "build it twice" guidance in Royce's initial waterfall-model article [14] and in Brooks' Mythical Man Month book [6], one must also add Brooks' "second system syndrome:" that developers, particularly for groupware, are likely to react over-ambitiously to experiences with initial prototypes or systems. This leads to the final major lesson learned:

Plan to develop more than two iterations of groupware systems

With persistence and focus on your mainstream end users, you can develop groupware systems, which both speed up the initial definition process and help stakeholders achieve a shared vision with lasting value across their application's entire life cycle.

7 References

- [1] Boehm B., Ross R., Theory W Software Project Management: Principles and Examples, IEEE Transactions on Software Engineering, July 1989, pp. 902-916.
- [2] Boehm B., Bose P., Horowitz E., Lee M.J., Software Requirements as Negotiated Win Conditions, Proceedings Intl. Conf. Rqts. Engineering, IEEE April 1994.
- [3] Boehm B., Egyed A., Kwan J., Port D., Shah A., Madachy R., Using the WinWin Spiral Model: A Case Study, IEEE Computer, 7:33-44, 1998.
- [4] Boehm B., Port D., Escaping the Software Tar Pit: Model Clashes and How to Avoid Them, Software Engineering Notes, Association for Computing Machinery, pp. 36-48, January 1999.
- [5] Boehm B., Grünbacher P., Supporting Collaborative Requirements Negotiation: The EasyWinWin Approach, International Conference on Virtual Worlds and Simulation, San Diego, SCS 2000.
- [6] Brooks F.P., The Mythical Man-Month, Reading, Mass: Addison-Wesley, 1975.
- [7] Covey S., The Seven Habits of Highly Effective People, Fireside Books, 1990.
- [8] Egyed A.F., Boehm B., Comparing Software System Requirements negotiation patterns, Journal for Systems Engineering, John Wiley & Sons, 1999.
- [9] The EasyWinWin Process Guide: USC-CSE and GROUPSYSTEMS.COM.
- [10] Fjermestad J., Hiltz S.R., An Assessment of Group Support Systems Research: Methodology, *Journal of Management Information Systems*, Winter/Spring 1998-99.
- [11] Gruenbacher P., Collaborative Requirements Negotiation with Easy WinWin, 2nd International Workshop on the Requirements Engineering Process, Greenwich London, IEEE Computer Society, 2000.
- [12] Harvey J. B., The Abilene Paradox and Other Meditations on Management (San Francisco: Jossey-Bass, 1988). The original publication of the Abilene Paradox appeared as: The Abilene Paradox: The Management of Agreement, in Organizational Dynamics, 1974.
- [13] Nunamaker, J. Briggs, R., Mittleman, D., Vogel, D., Balthazard, P., Lessons from a Dozen Years of Group Support Systems Research: A Discussion of Lab and Field Findings, Journal of Management Information Systems, Winter 1996-97, 13(3), 163-207.
- [14] Royce W.W., Managing the Development of Large Software Systems, Proceedings of IEEE WESCON, pp. 1-9, 1970.

- [15] Shepherd M.M., Briggs R.O., Reinig B.A., Yen J., Nunamaker J.F., Jr. Invoking Social Comparison to Improve Electronic Brainstorming: Beyond Anonymity, *Journal of Management Information Systems*. 12(3):155-170, 1995-96.
- [16] Waitley D., *The Double Win*, Berkeley Books, 1985.

Acknowledgments

This research is sponsored by DARPA through Rome Laboratory under contract number F30602-94-C-0195, by the Austrian Science Fund (Erwin Schrödinger Grant 1999/J 1764), and by the affiliates of the USC Center for Software Engineering: Aerospace, Automobile Club of Southern California, C-Bridge, Chung – Ang U. (Korea), Draper Labs, Draper Labs, Electronic Data Systems Corporation (EDS), Federal Aviation Administration (FAA), Fidelity, GDE Systems, Group Systems.Com, Hughes, Institute for Defense Analysis (IDA), Litton Industries, Inc., Lockheed Martin Corporation, Lucent Technologies, Microsoft, Motorola, Inc., Northrop Grumman Corporation, Rational Software Corporation, Raytheon/East, Raytheon/West, SAIC (Science Applications International Corporation), Software Engineering Institute (SEI Carnegie-Mellon University), Software Productivity Consortium (SPC), Sun Microsystems, Telcordia Technologies, The Boeing Company, TRW, Inc., U.S. Air Force Research Laboratory, U.S. Army Research Laboratory, US Army TACOM, Xerox Corporation.

We also thank the definers and developers of the first three versions of WinWin: Ellis Horowitz, Dan Port, Prasanta Bose, Yimin Bao, Anne Curran, Alex Egyed, Hoh In, Joo Lee, Jure Lee, Mingjune Lee, and Jungwon Park; and users of the four WinWin systems: Frank Beltz (TRW), Garry Brannum (Northrop Grumman), Walter Green (Aerospace), Elizabeth Kean (AFRL), Judy Kerner (Aerospace), Julie Kwan (USC), Andrew Landisman (TRW), Anne Lynch (USC), Ray Madachy (Litton), Azad Madni (Perceptronics), Nikunj Metha (USC and MediaConnex), Steve Mosher (USC), Karen Owens (Aerospace), Arnold Pittler (Motorola), and John Salasin (DARPA).

Trademarks

The following terms are trademarks of GroupSystems.com: thinkLet, BucketWalk, Could-Be-Should-Be, CrowBar, Free-Brainstorming, FastFocus, Point-Counterpoint, PopcornSort.



University of Southern California
Center for Software Engineering

Spiral Development: Experience, Principles, and Refinements

Barry Boehm, USC
Spiral Experience Workshop
February 9, 2000

boehm@sunset.usc.edu
<http://sunset.usc.edu/MBASE>

2/9/00

©USC-CSE

1

This presentation opened the USC-SEI Workshop on Spiral Development* Experience and Implementation Challenges held at USC February 9-11, 2000. The workshop brought together leading executives and practitioners with experience in transitioning to spiral development of software-intensive systems in the commercial, aerospace, and government sectors. Its objectives were to distill the participants' experiences into a set of critical success factors for transitioning to and successfully implementing spiral development, and to identify the most important needs, opportunities, and actions to expedite organizations' transition to successful spiral development.

To provide a starting point for addressing these objectives, I tried in this talk to distill my experiences in developing and transitioning the spiral model at TRW; in using it in system acquisitions at DARPA; in trying to refine it to address problems that people have had in applying it in numerous commercial, aerospace, and government contexts; and in working with the developers of major elaborations and refinements of the spiral model such that the Software Productivity Consortium's Evolutionary Spiral Process [SPC, 1994] and Rational, Inc's Rational Unified Process [Royce, 1998; Kruchten 1999; Jacobson et al., 1999]. I've modified the presentation somewhat to reflect the experience and discussions at the Workshop.

*For the workshop, "development" was defined to include life cycle evolution of software-intensive systems and such related practices as legacy system replacement and integration of commercial-off-the-shelf (COTS) components.

Spiral Model and MBASE

•Spiral experience

- Critical success factors

- Invariants and variants

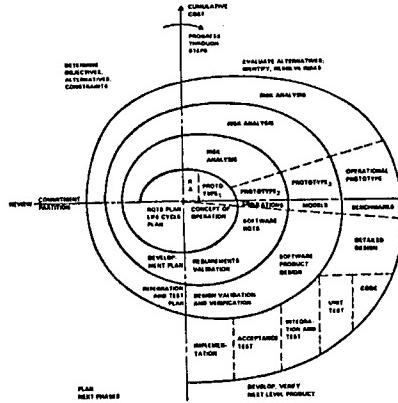
- Stud poker analogy

•Spiral refinements

- WinWin spiral

- Life cycle anchor points

- MBASE



2/9/00

©USC-CSE

2

This chart includes the original spiral model figure published in [Boehm, 1988]. It captures the major spiral model features: cyclic concurrent engineering; risk driven determination of process and product; growing a system via risk-driven experimentation and elaboration; and lowering development cost by early elimination of nonviable alternatives and rework avoidance. It indicates that the spiral model is actually a risk-driven process model generator, in which different risk patterns can lead a project to use evolutionary prototyping, waterfall, incremental, or other subsets of the process elements in spiral model diagram.

However, the chart contains some oversimplifications that have caused a number of misconceptions to propagate about the spiral model. These misconceptions may fit a few rare risk patterns, but are definitely not true for most risk patterns. The most significant misconceptions to avoid are: that the spiral is just a sequence of waterfall increments; that everything on the project follows a single spiral sequence; that every element in the diagram needs to be visited in the order indicated; and that there can be no backtracking to revisit previous decisions.

The presentation tries to clarify these points by presenting a set of spiral model critical success factors in terms of a set of invariants that should be true for every successful spiral implementation, and in terms of a set of variants that can be considered as alternative approaches to successful spiral implementation. This presentation also shows how the spiral model can be used for a more cost-effective incremental commitment of funds via an analogy of the spiral model to stud poker. It then presents some experience-based refinements of the spiral model developed to address spiral usage problems encountered over the years; the WinWin spiral model; a set of spiral-compatible life cycle anchor points; and the Model-Based (System) Architecting and Software Engineering (MBASE) approach. It concludes by summarizing some "hazardous-spiral look-alikes" to avoid, and by identifying a wide variety of projects which satisfied the spiral invariants and succeeded.

First, though, it begins with a simple overview definition to capture the essence of the spiral model.



“Spiral Development Model:” Candidate Definition

The spiral development model is a risk-driven process model generator. It is used to guide multi-stakeholder concurrent engineering of software-intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

A process model answers two main questions:

- What should the project do next?
- How long should the project continue doing it?

The spiral model holds that the answers to these questions vary from project to project, and that the variation is driven by risk considerations. It emphasizes the importance of having all of the project's success-critical stakeholders participate concurrently in defining and executing the project's processes (it uses risk considerations to ensure that progress is not overly slowed down by stakeholder overparticipation). It can be used to integrate software, hardware, and systems considerations, but is most important to use for software-intensive systems.

The cyclic nature of the spiral model was illustrated in Chart 2. The anchor-point stakeholder-commitment milestones are discussed in Charts 17-19.



Spiral Invariants and Variants - 1

- Critical success factor examples

Invariants	Why Invariant	Variants
1. Concurrent rather than sequential determination of artifacts (OCD, Rqts, Design, Code, Plans) in each spiral cycle.	<ul style="list-style-type: none"> Avoids premature sequential commitments to Rqts, Design, COTS, combination of cost/schedule performance <ul style="list-style-type: none"> - 1 sec. response time 	1a. Relative amount of each artifact developed in each cycle. 1b. Number of concurrent mini-cycles in each cycle.
2. Consideration in each cycle of critical-stakeholder objectives and constraints, product and process alternatives, risk identification and resolution, stakeholder review and commitment to proceed.	<ul style="list-style-type: none"> Avoids commitment to stakeholder-unacceptable or overly risky alternatives. Avoids wasted effort in elaborating unsatisfactory alternatives. <ul style="list-style-type: none"> - Mac-based COTS 	2a. Choice of risk resolution techniques: prototyping, simulation, modeling, benchmarking, reference checking, etc. 2b. Level of effort on each activity within each cycle.
3. Level of effort on each activity within each cycle driven by risk considerations.	<ul style="list-style-type: none"> Determines "how much is enough" of each activity: domain engr., prototyping, testing, CM, etc. <ul style="list-style-type: none"> - Pre-ship testing Avoids overkill or belated risk resolution. 	3a. Choice of methods used to pursue activities: MBASE/ WinWin, Rational USD, JAD, QFD, ESP, ... 3b. Degree of detail of artifacts produced in each cycle.

2/9/00

©USC-CSE

4

This chart summarizes the first three of the six identified spiral invariants:

1. Concurrent rather than sequential determination of artifacts.
2. Consideration in each spiral cycle of the main spiral elements: critical-stakeholder objectives and constraints; product and process alternatives; risk identification and resolution; stakeholder review and commitment to proceed.
3. Using risk considerations to determine the level of effort to be devoted on each activity within each spiral cycle.

Chart 13 summarizes the second three spiral invariants:

4. Using risk considerations to determine the degree of detail of each artifact produced in each spiral cycle.
5. Managing stakeholder life-cycle commitments via three Anchor Point milestones: Life Cycle Objectives (LCO), Life Cycle Architecture(LCA), and Initial Operational Capability(IOC).
6. Emphasis on system and life cycle activities and artifacts rather than software and initial development activities and artifacts.

Both this chart and Chart 13 summarize the invariants; critical-success-factor reasons why they are essential invariants, and associated optional variants. The next few charts elaborate each individual invariant.



Spiral Invariant 1: Concurrent Determination of Key Artifacts (Ops Concept, Rqts, Design, Code, Plans)

- Why invariant
 - Avoids premature sequential commitments to system requirements, design, COTS, combination of cost/schedule/ performance
 - 1 sec response time
- Variants
 - 1a. Relative amount of each artifact developed in each cycle.
 - 1b. Number of concurrent mini-cycles in each cycle.
- Models excluded
 - Incremental sequential waterfalls with high risk of violating waterfall model assumptions

Spiral invariant 1 states that it is success-critical to concurrently determine a compatible and feasible combination of key artifacts: the operational concept, the system and software requirements, the system and software architecture and design, key elements of code (COTS, reused components, prototypes, success-critical components or algorithms), and plans.

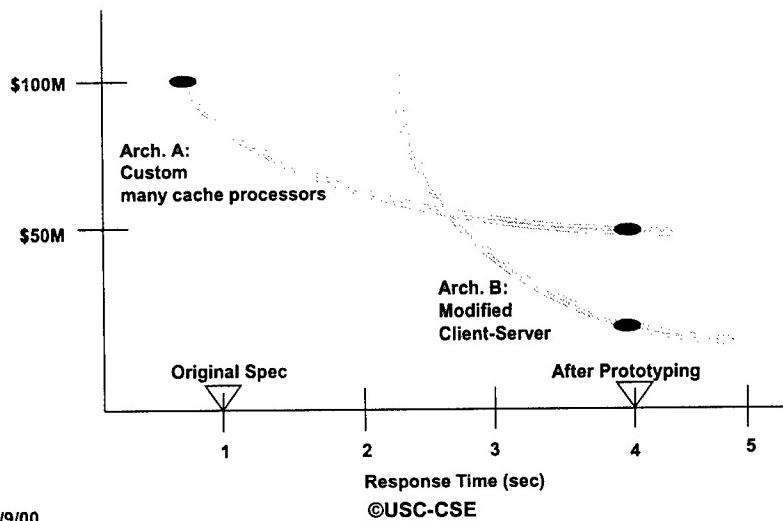
Why is this a success-critical invariant? Because sequential determination of the key artifacts will prematurely overconstrain, and often extinguish, the project's ability to develop a system which satisfies the stakeholders' essential success conditions. Examples are premature commitments to hardware platforms, to incompatible combinations of COTS components [Garlan et al., 1995], and to requirements whose achievability has not been validated. Chart 6 provides an example of the kinds of problems that occur when high-risk requirements are prematurely frozen.

The variants 1a and 1b indicate that the product and process internals of the concurrent engineering activity are not invariant. For a low technology, interoperability-critical system, the initial spiral products will be requirements-intensive. For a high-technology, more standalone system, the initial spiral products will be prototype code-intensive. Also, there is no invariant number of mini-cycles (e.g., individual prototypes for COTS, algorithm, or user-interface risks) within a given spiral cycle.

This invariant excludes one model often labeled as a spiral process, but which is actually a "hazardous spiral look-alike." This is the use of a sequence of incremental waterfall developments with a high risk of violating the underlying assumptions of the waterfall model described in Chart 7.



Sequential Engineering Buries Risk



In the early 1980s, a large government organization contracted with TRW to develop an ambitious information query and analysis system. The system would provide more than 1,000 users, spread across a large building complex, with powerful query and analysis capabilities for a large and dynamic database.

TRW and the customer specified the system using a classic sequential-engineering waterfall development model. Based largely on user need surveys and an oversimplified high-level performance analysis, they fixed into the contract a requirement for a system response time of less than one second.

Two thousand pages of requirements later, the software architects found that subsecond performance could only be provided via a highly customized design that attempted to anticipate query patterns and cache copies of data so that each user's likely data would be within one second's reach. The resulting hardware architecture had more than 25 super-midicomputers busy caching data according to algorithms whose actual performance defied easy analysis. The scope and complexity of the hardware-software architecture brought the estimated cost of the system to nearly \$100 million, driven primarily by the requirement for a one-second response time.

Faced with this unattractive prospect, the customer and developer decided to develop a prototype of the system's user interface and representative capabilities to test. The results showed that a four-second response time would satisfy users 90 percent of the time. A four-second response time dropped development costs closer to \$30 million [Boehm, 2000]. Thus, the premature specification of a 1-second response time buried the risk of creating an overexpensive and time-consuming system development.



Waterfall Model Assumptions

1. The requirements are knowable in advance of implementation.
2. The requirements have no unresolved, high-risk implications
 - e.g., risks due to COTS choices, cost, schedule, performance, safety, security, user interfaces, organizational impacts
3. The nature of the requirements will not change very much
 - During development; during evolution
4. The requirements are compatible with all the key system stakeholders' expectations
 - e.g., users, customer, developers, maintainers, investors
5. The right architecture for implementing the requirements is well understood.
6. There is enough calendar time to proceed sequentially.

2/9/00

©USC-CSE

7

This chart summarizes the assumptions about a software project's state of nature that need to be true for the waterfall model to succeed. If all of these are true, then it is a project risk not to specify the requirements, and the waterfall model becomes a risk-driven special case of the spiral model. If any of the assumptions are untrue, then specifying a complete set of requirements in advance of risk resolution will commit a project to assumptions/requirements mismatches that will lead the project into trouble.

Assumption 1 -- the requirements are knowable in advance of implementation -- is generally untrue for new user-interactive systems, because of the IKIWISI syndrome. When asked for their required screen layout for a new decision-support systems, users will generally say, "I can't tell you, but I'll know it when I see it (IKIWISI)." In such cases, a concurrent prototyping/requirements/architecture approach is needed.

The effects of invalidity in assumptions 2, 4, and 5 are well illustrated by the example in Chart 5. The 1-second response time requirement was unresolved and high-risk. It was compatible with the users' expectations, but not with the customer's budget expectations. And the need for an expensive custom architecture was not understood in advance.

The effects of invalidity in assumptions 3 and 6 are well illustrated by electronic commerce projects. There, the volatility of technology and the marketplace is so high that requirements and traceability updates will swamp the project in overhead. And the amount of initial calendar time it takes to work out a complete set of detailed requirements that are likely to change several times downstream is not a good investment of the scarce time to market available to develop an initial operational capability.



Spiral Invariant 2: Each cycle does objectives, constraints, alternatives, risks, review, commitment to proceed

- Why invariant
 - Avoids commitment to stakeholder-unacceptable or overly risky alternatives.
 - Avoids wasted effort in elaborating unsatisfactory alternatives.
 - Windows-only COTS
- Variants
 - 2a. Choice of risk resolution techniques: prototyping, simulation, modeling, benchmarking, reference checking, etc.
 - 2b. Level of effort on each activity within each cycle.
- Models excluded
 - Sequential phases with key stakeholders excluded

2/9/00

©USC-CSE

8

Spiral invariant 2 identifies the activities in each quadrant of the original spiral diagram which need to be done in each spiral cycle. These include consideration of critical-stakeholder objectives and constraints; elaboration and evaluation of project and process alternatives for achieving the objectives subject to the constraints; identification and resolution of risks attendant on choices of alternative solutions; and stakeholder's review and commitment to proceed based on satisfaction of their critical objectives and constraints.

If all of these are not considered, the project may prematurely commit itself to alternatives that are either unacceptable to key stakeholders or overly risky. Or it can waste a good deal of effort in elaborating an alternative that could have been shown earlier to be unsatisfactory. Chart 10 provides a representative example.

Spiral invariant 2 does not mandate particular generic choices of risk resolution techniques. However, there are risk management guidelines, e.g., [Boehm, 1989], that suggest best-candidate risk resolution techniques for the major sources of project risk. Invariant 2 also does not mandate particular levels of effort for the activities performed during each cycle. This means, for example, that software cost estimation models cannot be precise about the amount of effort and cost required for each cycle.

This invariant excludes another "hazardous spiral look-alike": organizing the project into sequential phases or cycles in which key stakeholders are excluded. Examples are excluding developers from system definition, excluding users from system construction, or excluding system maintainers from either definition or construction (see Chart 11).



Windows-Only COTS Example: Digital Library Artifact Viewer

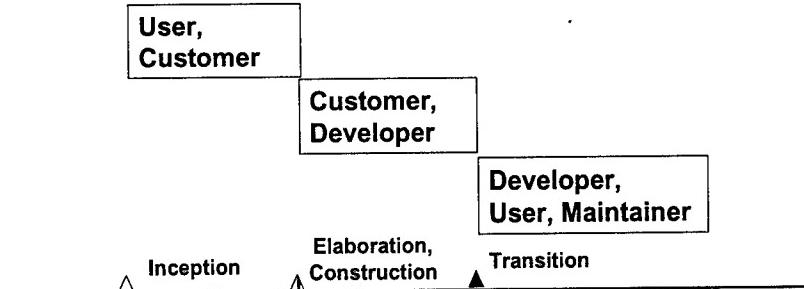
- Great prototype using ER Mapper
 - Tremendous resolution
 - Incremental-resolution artifact display
 - Powerful zoom and navigation features
- Only runs well on Windows
 - Mac, Unix user communities forced to wait
 - Overoptimistic assumptions on length of wait
- Eventual decision to drop ER Mapper

One of the current USC digital library projects is developing a web-based viewer for oversized artifacts (e.g., newspapers, large images). The initial prototype featured a tremendously powerful and high-speed viewing capability, based on a COTS product called ER Mapper. The initial project review approved selection of this COTS product, even though it only ran well on Windows platforms, and the Library had significant Macintosh and Unix user communities. This decision was based on initial indicators that Mac and Unix versions of ER Mapper would be available soon.

However, subsequent investigations indicated that it would be a long time before such Mac and Unix capabilities would become available. At a subsequent review, ER Mapper was dropped in favor of a less powerful but fully portable COTS product, Mr. SID, but only after a good deal of wasted effort was devoted to elaborating the ER Mapper solution. If a representative of the Mac or UNIX user community had been involved in the early project decisions, the homework leading to choosing Mr. SID would have been done earlier, and the wasted effort in elaborating the ER Mapper solution would have been avoided.



Models Excluded: Sequential Phases Without Key Stakeholders



- High risk of win-lose even with spiral phases
 - Win-lose evolves into lose-lose
- Key criteria for IPT members (AFI 63-123)
 - Representative, empowered, knowledgeable, collaborative, committed

2/9/00

©USC-CSE

10

Even though the phases shown in this chart may look like risk-driven spiral cycles, this spiral look-alike will be hazardous because its exclusion of key stakeholders is likely to cause critical risks to go undetected. Excluding developer participation in early cycles can lead to project commitments based on unrealistic assumptions about developer capabilities. Excluding users or maintainers from development cycles can lead to win-lose situations, which generally evolve into lose-lose situations [Boehm-Ross, 1989].

Projects must also guard against having the appearance but not the reality of stakeholder participation by accepting an unqualified member of an integrated product team (IPT). A good set of criteria for qualified IPT members described in [Boehm et al., 1998] and adopted in [USAF, 2000] is to ensure that IPT members are representative (of organizational rather than personal positions); empowered (to make commitments which will be honored by their organizations); knowledgeable (of their organization's critical success factors); collaborative, and committed.



Spiral Invariant 3: Level of Effort Driven by Risk Considerations

- **Why invariant**
 - Determines ‘how much is enough’ of each activity:
domain engr., prototyping, testing, CM, etc.
 - Pre-ship testing
 - Avoids overkill or belated risk resolution.
- **Variants**
 - 3a. Choice of methods used to pursue activities:
MBASE/WinWin, Rational RUP, JAD, QFD, ESP, ...
 - 3b. Degree of detail of artifacts produced in each cycle.
- **Models excluded**
 - Risk-insensitive evolutionary or incremental development

2/9/00

©USC-CSE

11

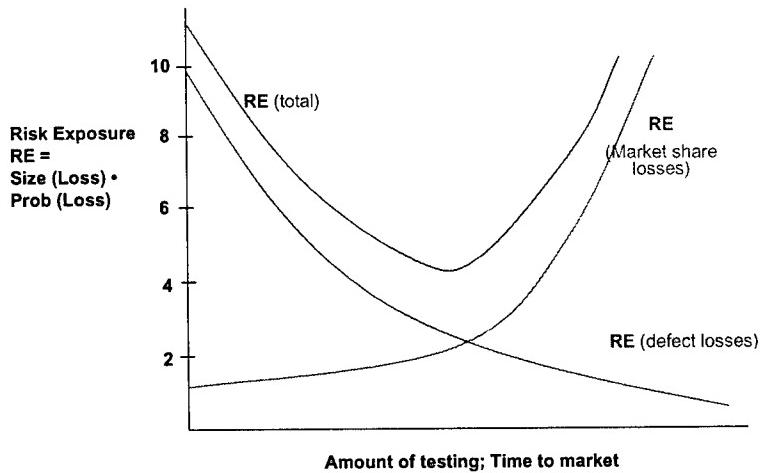
Spiral invariant 3 uses risk considerations to provide answers to one of the most difficult questions for a project to answer: how much of a given activity (domain engineering, prototyping, testing, configuration management, etc.) is enough? An example of how this works for testing is provided in Chart 12. It shows that if you plot a project's risk exposure as a function of time spent testing, there is a point at which risk exposure is minimized. Spending significantly more testing time than this is an overkill leading to late market entry and decreased market capitalization. Spending significantly less testing time than this is an underkill leading to early market entry with products that are so unreliable that the company loses market share and market capitalization.

Given that risk profiles vary from project to project, this means that the risk-minimizing level of testing effort will vary from project to project. The amount of effort devoted to other activities will also vary as a function of a project's risk profile, again presenting a challenge for software cost models' ability to estimate a project's effort distribution by activity and phase. Another variant is an organization's choice of particular methods for risk assessment and management.

Hazardous spiral model look-alikes excluded by invariant 3 are risk-insensitive evolutionary development (e.g., neglecting scalability risks) or risk-insensitive incremental development (e.g., suboptimizing on increment 1 with a point-solution architecture which must be dropped or heavily reworked to accommodate future increments); or impeccable spiral plans with no commitment to managing the risks identified.



Pre-Ship Test Risk Exposure



2/9/00

©USC-CSE

12

This chart shows how risk considerations can help determine "how much testing is enough" before shipping a product. This can be determined by adding up the two main sources of risk Exposure, $RE = \text{Probability (Loss)} \cdot \text{Size (Loss)}$, incurred by two sources of loss: loss of profitability due to product defects, and loss of profitability due to delays in capturing market share. The more testing the project does, the lower becomes the risk exposure due to defects, as discovered defects reduce both the size of loss due to defects and the probability that undiscovered defects still remain. However, the more time the project spends testing, the higher are both the probability of loss due to competitors entering market and the size of loss due to decreased profitability on the remaining market share.

As shown in Chart 12, the sum of these risk exposures achieves a minimum at some intermediate level of testing. The location of this minimum-risk point in time will vary by type of organization. For example, it will be considerably shorter for a ".com" company than it will for a safety-critical product such as a nuclear powerplant. Calculating the risk exposures also requires an organization to accumulate a fair amount of calibrated experience on the probabilities and size of losses as functions of test duration and delay in market entry.



Spiral Invariants and Variants - 2

Invariants	Why Invariant	Variants
4. Degree of detail of artifacts produced in each cycle driven by risk considerations.	<ul style="list-style-type: none"> Determines "how much is enough" of each artifact (OCD, Rqts, Design, Code, Plans) in each cycle. Avoids overkill or belated risk resolution 	4a. Choice of artifact representations (SA/SD, UML, MBASE, formal specs, programming languages, etc.)
5. Managing stakeholder life-cycle commitments via the LCO, LCA, and IOC Anchor Point milestones (getting engaged, getting married, having your first child),	<ul style="list-style-type: none"> Avoids analysis paralysis, unrealistic expectations, requirements creep, architectural drift, COTS shortfalls and incompatibilities, unsustainable architectures, traumatic cutovers, useless systems. 	5a. Number of spiral cycles or increments between anchor points. 5b. Situation-specific merging of anchor point milestones.
6. Emphasis on system and life cycle activities and artifacts rather than software and initial development activities and artifacts.	<ul style="list-style-type: none"> Avoids premature suboptimization on hardware, software, or initial development considerations. 	6a. Relative amount of hardware and software determined in each cycle. 6b. Relative amount of capability in each life cycle increment. 6c. Degree of productization (alpha, beta, shrink-wrap, etc.) of each life cycle increment.

2/9/00

©USC-CSE

13

This chart summarizes the second three spiral invariants:

4. Using risk considerations to determine the degree of detail of each artifact produced in each spiral cycle.
5. Managing stakeholder life-cycle commitments via three Anchor Point milestones: Life Cycle Objectives (LCO), Life Cycle Architecture (LCA), and Initial Operational Capability (IOC).
6. Emphasis on system and life cycle activities and artifacts rather than software and initial development activities and artifacts.

Both this chart and Chart 13 summarize the invariants, critical-success-factor reasons why they are essential invariants, and associated optional variants. The next few charts elaborate the second three invariants.



Spiral Invariant 4: Degree of Detail Driven by Risk Considerations

- Why invariant
 - Determines “how much is enough” of each artifact (OCD, Rqts, Design, Code, Plans) in each cycle.
 - Screen layout rqts.
 - Avoids overkill or belated risk resolution.
- Variants
 - 4a. Choice of artifact representations (SA/SD, UML, MBASE, formal specs, programming languages, etc.)
- Models excluded
 - Complete, consistent, traceable, testable requirements specification for systems involving significant levels of GUI, COTS, or deferred decisions

Spiral invariant 4 is the product counterpart of invariant 3: that risk considerations determine the degree of detail of products as well as processes. This means, for example, that the traditional ideal of a complete, consistent, traceable, testable requirements specification is not a good idea for a number of product components, such as a graphic user interface (GUI). Here, the risk of precisely specifying screen layouts in advance of development involves a high probability of locking an awkward user interface into the development contract, while the risk of not specifying screen layouts is low, given the general availability of flexible GUI-builder tools (see Chart 15). Even aiming for full consistency and testability can be risky, as it creates a pressure to prematurely specify decisions that would better be deferred (e.g., the form and content of exception reports). However, as indicated in Chart 15, some risk patterns make it very important to have precise specifications.

Related spiral variants are the project's choices of representations for product artifacts.



Risk-Driven Specifications

- If it's risky not to specify precisely, Do
 - Hardware-software interface
 - Prime-subcontractor interface
- If it's risky to specify precisely, Don't
 - GUI layout
 - COTS behavior

This chart gives further examples of when it is risky to overspecify software features (GUI layouts, COTS behavior) and when it is risky to underspecify them (critical interfaces with hardware or with externally developed software).



Spiral Invariant 5: Use of LCO, LCA, IOC, Anchor Point Milestones

- Why invariant

- Avoids analysis paralysis, unrealistic expectations, requirements creep, architectural drift, COTS shortfalls and incompatibilities, unsustainable architectures, traumatic cutovers, useless systems.

- Variants

- 5a. Number of spiral cycles or increments between anchor points.
 - 5b. Situation-specific merging of anchor point milestones
 - Can merge LCO and LCA when adopting an architecture from mature 4GL, product line

- Models excluded

- Evolutionary or incremental development with no life cycle architecture

2/9/00

©USC-CSE

16

A major difficulty of the original spiral model was its lack of intermediate milestones to serve as commitment points and progress checkpoints [Forsberg et al., 1996]. This difficulty has been remedied by the development of a set of anchor point milestones: Life Cycle Objectives (LCO), Life Cycle Architecture (LCA), and Initial Operational Capability (IOC) [Boehm, 1996].

Chart 17 describes the role of the LCO, LCA, and IOC milestones as stakeholder commitment points in the software life cycle. Chart 18 provides details on the content and pass/fail criteria for the LCO and LCA milestones. Chart 19 summarizes the content of the IOC milestone.

Appropriate variants include the number of spiral cycles of development increments between the anchor points. In some cases, anchor point milestones can be merged. In particular, a project deciding to use a mature and appropriately scalable fourth generation language (4GL) or product line framework will have already determined its choice of life cycle architecture by its LCO milestone, enabling the LCO and LCA milestones to be merged.

The LCA milestone is particularly important, as its pass/fail criteria enable stakeholders to hold up projects attempting to proceed into evolutionary or incremental development without a life cycle architecture. Chart 20 summarizes other evolutionary development assumptions whose validity should be verified at the LCA milestone.

Charts 21-25 summarize other aspects of the spiral model relevant to the anchor point milestones, such as their support of incremental commitment and their relation to the Rational Unified Process [Royce, 1998; Kruchten, 1998; Jacobson et al., 1999] and the USC MBASE approach [Boehm-Port, 1999a; Boehm-Port, 1999b; Boehm et al., 2000].



Life Cycle Anchor Points

- Common System/Software stakeholder commitment points
 - Defined in concert with Government, industry affiliates
 - Coordinated with the Rational Unified Process
- Life Cycle Objectives (LCO)
 - Stakeholders' commitment to support architecting
 - Like getting engaged
- Life Cycle Architecture (LCA)
 - Stakeholders' commitment to support full life cycle
 - Like getting married
- Initial Operational Capability (IOC)
 - Stakeholders' commitment to support operations
 - Like having first child

2/9/00

©USC-CSE

17

The anchor point milestones were defined in a pair of USC Center for Software Engineering Affiliates' workshops, originally for the purpose of defining a set of common reference points for COCOMO II cost model estimates of spiral model projects' cost and schedule. One of the Affiliates, Rational, Inc., had been defining the phases of its Rational Unified Process, and adopted the anchor point milestones as its phase gates.

The first two anchor points are the Life Cycle Objectives (LCO) and Life Cycle Architecture (LCA). At each of these anchor points the key stakeholders review six artifacts: *operational concept description, prototyping results, requirements description, architecture description, life cycle plan, and feasibility rationale* (see next chart for details).

The feasibility rationale covers the key pass/fail question: "If I build this product using the specified architecture and processes, will it support the operational concept, realize the prototyping results, satisfy the requirements, and finish within the budgets and schedules in the plan?" If not, the package should be reworked.

The focus of the LCO review is to ensure that at least one architecture choice is viable from a business perspective. The focus of the LCA review is to commit to a single detailed definition of the review artifacts. The project must have either eliminated all significant risks or put in place an acceptable risk-management plan.

The LCO milestone is the equivalent of getting engaged, and the LCA milestone is the equivalent of getting married. As in life, if you marry your architecture in haste, you and your stakeholders will repent at leisure. The third anchor point milestone, the Initial Operational Capability (IOC), constitutes an even larger commitment: It is the equivalent of having your first child.



Win Win Spiral Anchor Points

(Risk-driven level of detail for each element)

Milestone Element	Life Cycle Objectives (LCO)	Life Cycle Architecture (LCA)
Definition of Operational Concept	<ul style="list-style-type: none"> - Top-level system objectives and scope - System boundaries - Environment parameters and assumptions - Evolution parameters - Operational concept - Operations and maintenance scenarios and parameters - Organizational life-cycle responsibilities (stakeholders) 	<ul style="list-style-type: none"> - Elaboration of system objectives and scope of increment - Elaboration of operational concept by increment
System Prototype(s)	<ul style="list-style-type: none"> - Exercise key usage scenarios - Resolve critical risks 	<ul style="list-style-type: none"> - Exercise range of usage scenarios - Resolve major outstanding risks
Definition of System Requirements	<ul style="list-style-type: none"> - Top-level functions, interfaces, quality attribute levels, including: <ul style="list-style-type: none"> - Growth vectors and priorities - Prototypes - Stakeholders' concurrence on essentials 	<ul style="list-style-type: none"> - Elaboration of functions, interfaces, quality attributes, and prototypes by increment - Identification of TBD's (to-be-determined items) - Stakeholders' concurrence on their priority concerns
Definition of System and Software Architecture	<ul style="list-style-type: none"> - Top-level definition of at least one feasible architecture - Physical and logical elements and relationships - Choices of COTS and reusable software elements - Identification of infeasible architecture options 	<ul style="list-style-type: none"> - Choice of architecture and elaboration by increment - Physical and logical components, connectors, configurations, constraints - COTS, reuse choices - Domain-architecture and architectural style choices - Architecture evolution parameters
Definition of Life-Cycle Plan	<ul style="list-style-type: none"> - Identification of life-cycle stakeholders <ul style="list-style-type: none"> - Users, customers, developers, maintainers, interoperators, general public, others - Identification of life-cycle process model <ul style="list-style-type: none"> - Top-level stages, increments - Top-level WWWWWWH[†] by stage 	<ul style="list-style-type: none"> - Elaboration of WWWWWWH[†] for initial Operational Capability (OC) - Partial elaboration, identification of key TBD's for later increments
Feasibility Rationale	<ul style="list-style-type: none"> - Assurance of consistency among elements above - via analysis, measurement, prototyping, simulation, etc. - Business case analysis for requirements, feasible architectures 	<ul style="list-style-type: none"> - Assurance of consistency among elements above - All major risks resolved or covered by risk management plan

*WWWWWWH: Why, What, When, Who, Where, How, How Much

2/9/00

©USC-CSE

18

Here are the major features of the LCO and LCA milestones which distinguish them from most current software milestones, which provide a rationale for their success-criticality on projects, and which enable them to function successfully as anchor points across many types of software development.

- Their focus is not on requirements snapshots or architecture point solutions, but on requirements and architectural specifications which anticipate and accommodate system evolution. This is the reason for calling them the "Life Cycle" Objectives and Architecture milestones.
- Elements can be either specifications or executing programs with data (e.g., prototypes, COTS products).
- The Feasibility Rationale is an essential element rather than an optional add-on.
- Stakeholder concurrence on the milestone elements is essential. This establishes mutual stakeholder buy-in to the plans and specifications, and enables a collaborative team approach to unanticipated setbacks rather than an adversarial approach as in most contract models.

A feature distinguishing the LCA milestone from the LCO milestone is the need to have all of the system's major risks resolved, or at least covered by an element of the system's risk management plan. For large systems, passing the LCA milestone is the point at which the project will significantly escalate at its staff level and resource commitments. Proceeding into this stage with major risks unaddressed has led to disasters for many large projects. Some good guidelines for software risk assessment can be found in [Boehm, 1989; Charette, 1989; Carr et al., 1993; and Hall, 1998].

A key feature of the LCO milestone is the need for the Feasibility Rationale to demonstrate a viable business case for the proposed system. Not only should this business case be kept up to date, but also it should be used as a basis for verifying that expected benefits will actually be realized (see chart 27).



Initial Operational Capability (IOC)

- **Software preparation**
 - Operational and support software
 - Data preparation, COTS licenses
 - Operational readiness testing
- **Site preparation**
 - Facilities, equipment, supplies, vendor support
- **User, operator, and maintainer preparation**
 - Selection, teambuilding, training

Another distinguishing feature of the LCO and LCA milestones is that they are the milestones with the most serious consequences if one gets any parts of them wrong. At the other end of the development cycle, the milestone with the most serious consequences of getting things wrong is the Initial Operational Capability (IOC). Greeting users with a new system having ill-matched software, poor site preparation, or poor users preparation has been a frequent source of user alienation and killed projects.

The key elements of the IOC milestone are:

- Software preparation, including both operational and support software with appropriate commentary and documentation; data preparation or conversion; the necessary licenses and rights for COTS and reused software, and appropriate operational readiness testing.
- Site preparation, including facilities, equipment, supplies, and COTS vendor support arrangements.
- User, operator and maintainer preparation, including selection, teambuilding, training and other qualification for familiarization, usage, operations, or maintenance.

As discussed on Chart 12, the nature of the IOC milestone is also risk-driven with respect to the system objectives determined in the LCO and LCA milestones. Thus, for example, these objectives drive the tradeoff between IOC date and quality of the product (e.g. between the safety-critical Space Shuttle Software and a market window-critical commercial software product). The difference between these two cases is narrowing as commercial vendors and users increasingly appreciate the market risks involved in buggy products [Cusumano-Selby, 1995].



Evolutionary Development Assumptions

- 1. The initial release is sufficiently satisfactory to key system stakeholders that they will continue to participate in its evolution.**
- 2. The architecture of the initial release is scalable to accommodate the full set of system life cycle requirements (e.g., performance, safety, security, distribution, localization).**
- 3. The operational user organizations are sufficiently flexible to adapt to the pace of system evolution**
- 4. The dimensions of system evolution are compatible with the dimensions of evolving-out the legacy systems it is replacing.**

2/9/00

©USC-CSE

20

All too often, an evolutionary development will start off with a statement such as, "We're not sure what to build, so let's throw together a prototype and evolve it until the users are satisfied." This approach is insensitive to several risks corresponding to the set of assumptions for a successful evolutionary development summarized in Chart 20.

Without some initial attention to user needs, the prototype may be so far from the users' needs that they consider it a waste of time to continue. As discussed on Chart 16, it will be risky to proceed without a life cycle architecture to support evolution. Another risk is "information sclerosis": the propensity for organizations to lock into operational procedures making it difficult to evolve toward better capabilities [Boehm, 1988]. A final frequent risk is that legacy systems are often too inflexible to adapt to desired directions of evolution. In such cases, a preferable process model is incremental development, with the increments determined by the ease of evolving-out portions of the legacy system to be replaced.



Spiral Model and Incremental Commitment: Stud Poker Analogy

- **Evaluate alternative courses of action**
 - Fold: save resources for other deals
 - Ante: buy at least one more round
- **Using incomplete information**
 - Hole cards: competitive situation
 - Rest of deck: chance of getting winner
- **Anticipating future possibilities**
 - Likelihood that next round will clarify outcome
- **Commit incrementally rather than all at once**
 - Challenge: DoD POM process makes this hard to do

2/9/00

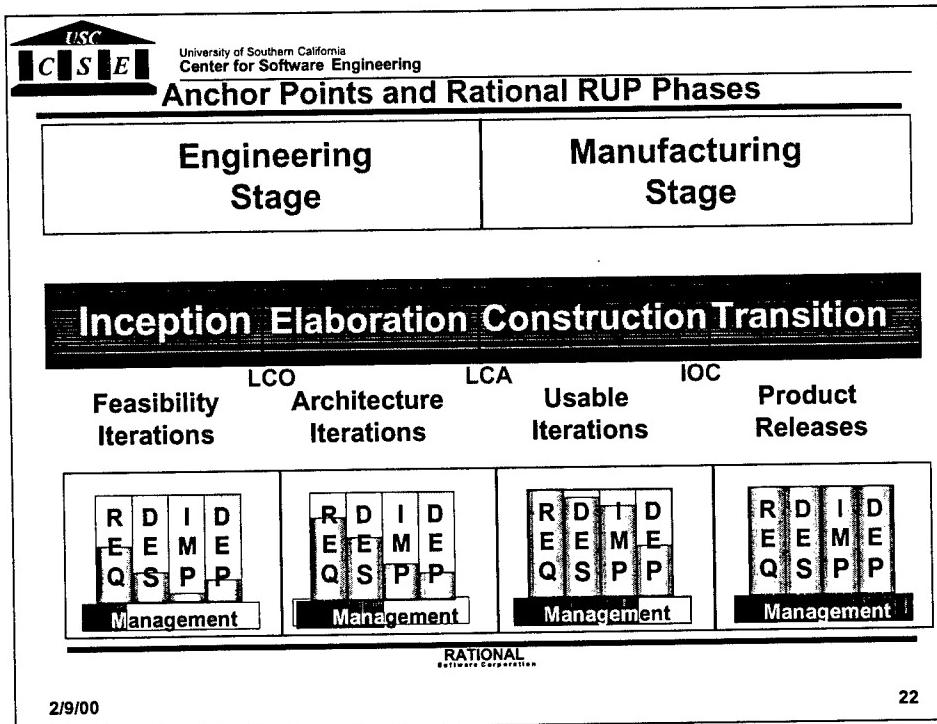
©USC-CSE

21

A valuable aspect of the original application of the spiral model to the TRW Software Productivity System was its ability to support incremental commitment of corporate resources to the exploration, definition, and development of the system, rather than requiring a large outlay of resources to the project before its success prospects were well understood [Boehm, 1988].

Funding a spiral development can thus be envisioned in a way similar to the game of stud poker. You can put a couple of chips in the pot and receive two hidden cards and one exposed card, along with the other players in the game. If your cards don't promise a winning outcome, you can drop out without a great loss. If your two hidden cards are both aces, you will probably bet on your prospects aggressively (although perhaps less so if you can see the other two aces as other players' exposed cards). In any case, you can decide during each round whether it's worth putting more chips in the pot to buy more information about your prospects for a win or whether it's better not to pursue this particular deal, based on the information available.

One of the main challenges for organizations such as the Department of Defense (DoD), is to find incremental commitment alternatives to its current Program Objectives Memorandum (POM) process which involves committing to the full funding of a program based on very incomplete early information.



Versions of this chart are in the three main books on the Rational Unified Process (RUP) [Royce, 1998; Kruchten, 1998; and Jacobson, et al., 1999]. It shows the relations between LCO, LCA, and IOC milestones and the RUP Inception, Elaboration, Construction, and Transition phases. It also illustrates that the requirements, design, implementation, and deployment artifacts are incrementally grown throughout the phases. As indicated in Variant 3b on Chart 11, the size of the shaded bars will vary from project to project.



Spiral Model Refinements

- Where do objectives, constraints, alternatives come from?

– Win Win extensions

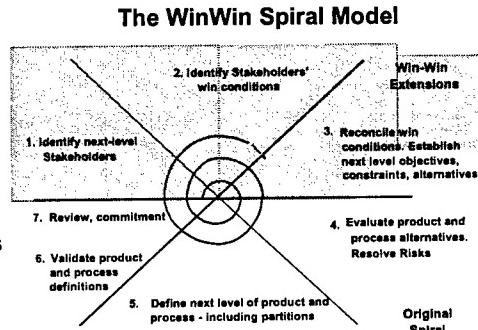
- Lack of intermediate milestones

– Anchor Points: LCO, LCA, IOC

– Concurrent-engineering spirals between anchor points

- Need to avoid model clashes, provide more specific guidance

– MBASE



2/9/00

©USC-CSE

23

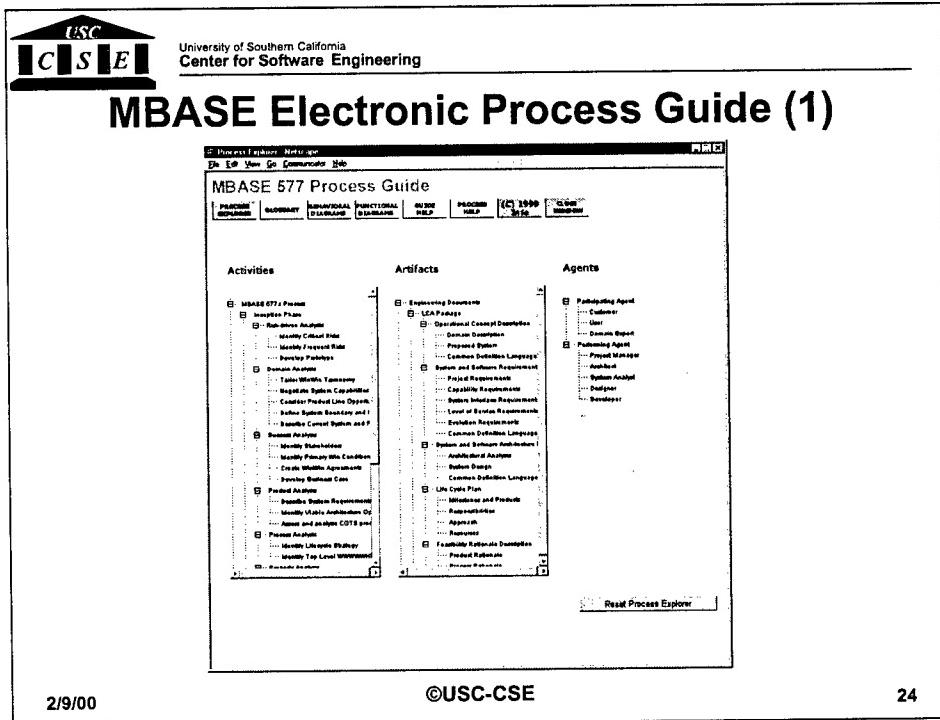
The original spiral model [Boehm, 1988] began each cycle of the spiral by performing the next level of elaboration of the prospective system's objectives, constraints and alternatives. A primary difficulty in applying the spiral model has been the lack of explicit process guidance in determining these objectives, constraints, and alternatives. The Win-Win Spiral Model [Boehm-Bose, 1994] uses the Theory W (win-win) approach [Boehm-Ross, 1989] to converge on a system's next-level objectives, constraints, and alternatives. This Theory W approach involves identifying the system's stakeholders and their win conditions, and using negotiation processes to determine a mutually satisfactory set of objectives, constraints, and alternatives for the stakeholders.

In particular, the nine-step Theory W process translates into the following Spiral Model extensions:

- Determine Objectives. Identify the system life-cycle stakeholders and their win conditions. Establish initial system boundaries and external interfaces.
- Determine Constraints. Determine the conditions under which the system would produce win-lose or lose-lose outcomes for some stakeholders.
- Identify and Evaluate Alternatives. Solicit suggestions from stakeholders. Evaluate them with respect to stakeholders' win conditions. Synthesize and negotiate candidate win-win alternatives. Analyze, assess, and resolve win-lose or lose-lose risks.
- Record Commitments, and areas to be left flexible, in the project's design record and life cycle plans.

Cycle Through the Spiral. Elaborate win conditions, evaluate and screen alternatives, resolve risks, accumulate appropriate commitments, and develop and execute downstream plans.

A further extension, the Model-Based (System) Architecting and Software Engineering (MBASE) approach, [Boehm-Port, 1999a; Boehm-Port, 1999b], provides more detailed definitions of the anchor point milestone elements [Boehm et al., 2000], and a process guide for deriving them (see next charts).



The MBASE Electronic Process Guide [Mehta, 1999] was developed using the SEI's Electronic Process Guide support tool [Kellner et al., 1999]. It uses Microsoft Access to store the process elements, using an Activities-Artifacts-Agents model, and translates the results into hyperlinked html for web-based usage. Thus, for example, when a user clicks on the "MBASE 577a Process" activity and the "Operational Concept Definition" artifact, the tool displays the corresponding elements as shown in Chart 25.

These elements are also hyperlinked. Thus, for example, a user can access a template for composing a project's Operational Concept Description by clicking on the "Templates" entry in its left column.

USC
C S E

University of Southern California
Center for Software Engineering

MBASE Electronic Process Guide (2)

MBASE 577 Process Guide

Activities:

- [] Activities
- [] Requirements Plan
- [] Business Plan
- [] System Requirements Plan
- [] Workbooks for MBASE

MBASE 577a Process

Overview:
The process followed by students of CS 577a for Digital Library projects

Purpose:
The objectives for the MBASE 577a process are:
 1. To define the basic process for the students of CS 577a for use in Digital Library and similar projects.
 2. To provide the students with important insights about the interdependence of MBASE process elements.

Decomposition:
The activity MBASE 577a Process is decomposed into the following:
 • [] Overview
 • [] Requirements Plan
 • [] Business Plan
 • [] System Requirements Plan
 • [] Workbooks for MBASE

Description:
The MBASE 577 process is a variant of the MBASE framework used by students in the USC CSE 577 course. It consists of four phases: Initiation, Planning, Execution, and Transition. Each phase is subdivided into several activities. The process is iterative, allowing a return to previous phases if necessary.

Tools and Techniques:
None

Pitfalls:
None

MBASE 577 Process Guide

Artifacts:

- [] Overview
- [] Requirements Plan
- [] Business Plan
- [] System Requirements Plan
- [] Workbooks for MBASE

Operations/ Concept Description

Overview:
Provide the overall context of the processed system and its operational concept

Purpose:
 • Describe the overall context of the system to be developed, and key requirements, needs, and constraints where the system is developed from.
 • Define the relationships of the system to be developed to its environment.
 • Define the system's operational concept, including the system's purpose, "Assumptions", how the system will work or produce since it is developed from the system's environment, and how the system will interact with its current operational context to the new operational context, and to establish the value of developing the new system.

Owner:
The artifact Operations/ Concept Description is owned by the agent System Analyst.

Decomposition:
The artifact Operations/ Concept Description is decomposed into the following:
 • [] Domain Description
 • [] Requirements Plan
 • [] Context Definition Language for Domain

Description:
OOD Audience and Participants

- [] Audience
- [] Customer for Domain Description
- [] Customer for Requirements Plan
- [] User Language and Define OOD appropriately for intended audience
- [] Requirements
- [] Behavior Model
- [] Behavior Model
- [] Template
- [] Summary

OOD High-Level Dependencies:

- [] System Requirements Plan
- [] System Requirements
- [] Design Plan
- [] Design Plan
- [] Detailed Description
- [] Detailed Description
- [] OOD Model
- [] Project Plan and Quality Plan for SISAD
- [] Detailed Description and Detailed Description for SISAC
- [] Detailed Description and Detailed Description for SISB

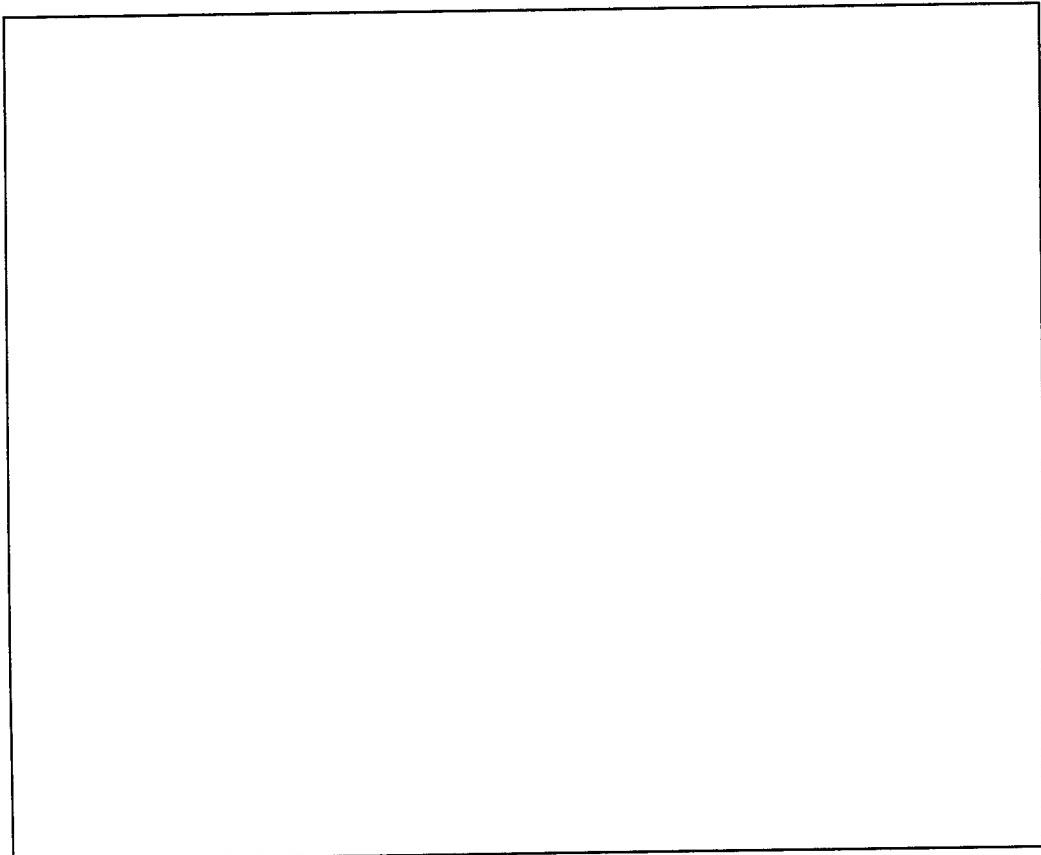
OOD Low-Level Dependencies:

- [] System Requirements
- [] System Requirements
- [] Design Plan
- [] Design Plan
- [] Detailed Description
- [] Detailed Description
- [] OOD Model
- [] Project Plan and Quality Plan for SISAD
- [] Detailed Description and Detailed Description for SISAC
- [] Detailed Description and Detailed Description for SISB

2/9/00

©USC-CSE

25





Spiral Invariant 6: Emphasis on System and Life Cycle Activities and Artifacts

- **Why invariant**

- Avoids premature suboptimization on hardware, software, or development considerations.
 - Scientific American

- **Variants**

- 6a. Relative amount of hardware and software determined in each cycle.
- 6b. Relative amount of capability in each life cycle increment
- 6c. Degree of productization (alpha, beta, shrink-wrap, etc.) of each life cycle increment.

- **Models excluded**

- Purely logical object-oriented methods
 - Insensitive to operational, performance, cost risks

2/9/00

©USC-CSE

26

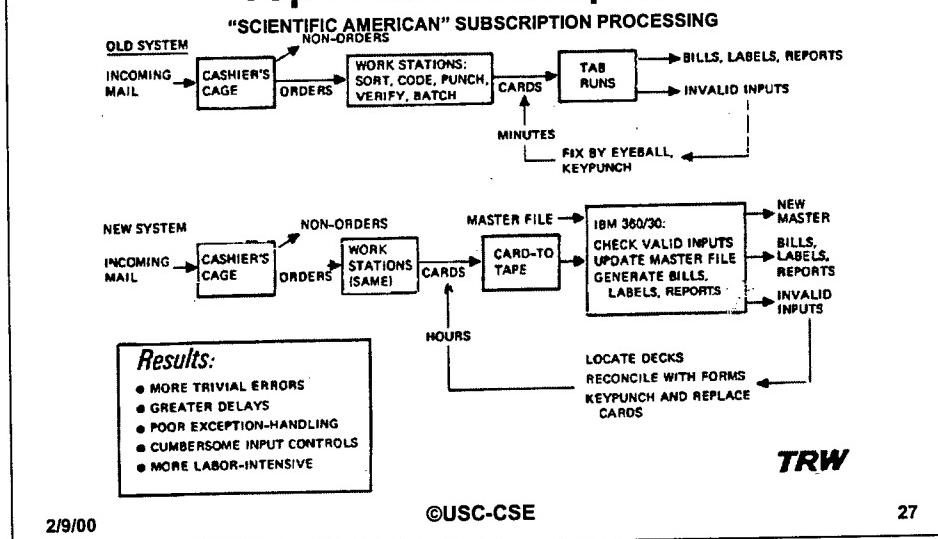
Spiral invariant 6 emphasizes that spiral development of software-intensive systems needs to focus not just on software construction aspects, but also on overall system and life cycle concerns. Software developers are particularly apt to fall into the oft-cited trap: "If your only tool is a hammer, the world begins to look like a collection of nails."

A good example is the Scientific American case study shown in the next chart. The software people looked for the part of the problem with a software solution (their "nail"), pounded it in with their software hammer, and left Scientific American worse off than when they started.

The spiral model's emphasis on using stakeholder objectives to drive system solutions, and on the life cycle anchor point milestones, guides projects to focus on system and life cycle concerns. Its use of risk considerations to drive solutions enables projects to tailor each spiral cycle to whatever mix of software and hardware, choice of capabilities, or degree of productization is appropriate.

Models excluded by invariant 6 include most published object-oriented analysis and design (OOA&D) methods, which are usually presented as abstract logical exercises independent of system performance or economic concerns. For example, in a recent survey of 16 OOA&D books, only 6 had the word "performance" in its index, and only 2 had the word "cost" in its index.

Problems With Programming-Oriented Top-Down Development



2/9/00

©USC-CSE

27

Scientific American's objectives were to reduce their subscription processing system's costs, errors, and delays. Rather than analyze the system's sources of costs, errors, and delays, the software house jumped in and focused on the part of the problem having a software solution. The result was a batch-processing computer system whose long delays put extra strain on the clerical portion of the system which had been the major source of the costs, errors, and delays in the first place. As seen in the chart, the business outcome was a new system with more errors, greater delays, higher costs, and less attractive work than its predecessor [Boehm, 1981].

This kind of outcome would have happened even if the software automating the tabulator-machine functions had been developed in a risk-driven cyclic approach. However, its Life Cycle Objectives milestone package would have failed its feasibility review, as it had no system-level business case demonstrating that the development of the software would lead to the desired reduction in costs, errors, and delays.

Had a thorough business case analysis been done, it would have identified the need to re-engineer the clerical business processes as well as to automate the manual tab runs. Further, as shown by recent methods such as the DMR Benefits Realization Approach [Thorp, 1998], the business case could have been used to monitor the actual realization of the expected benefits, and to apply corrective action to either the business process re-engineering or the software engineering portions of the solution (or both) as appropriate.



Summary: Hazardous Spiral Look-Alikes

- Incremental sequential waterfalls with significant COTS, user interface, or technology risks
- Sequential spiral phases with key stakeholders excluded from phases
- Risk-insensitive evolutionary or incremental development
- Evolutionary development with no life-cycle architecture
- Insistence on complete specs for COTS, user interface, or deferred-decision situations
- Purely logical object-oriented methods with operational, performance, or cost risks
- Impeccable spiral plan with no commitment to managing risks

As with many methods, the spiral model has a number of “hazardous look-alikes,” which have been discussed in the previous charts.

Incremental sequential waterfalls with significant COTS, user interface, or technology risks are discussed in Charts 6 and 7. Sequential spiral phases with key stakeholders excluded from phases are discussed in Charts 8, 9, and 10. Risk-insensitive evolutionary or incremental development is discussed in Charts 11, 16, and 20, as is evolutionary development with no life-cycle architecture. Insistence on complete specs for COTS, user interface, or deferred-decision situations is discussed in Charts 14 and 15. Purely logical object-oriented methods with operational, performance, or cost risks are discussed in Chart 26. Impeccable spiral plans with no commitment to managing risks are discussed in Charts 11 and 23.



Summary: Successful Spiral Examples

- **Rapid commercial:** C-Bridge's RAPID process
- **Large commercial:** AT&T/Lucent/Telcordia spiral extensions
- **Commercial hardware-software:** Xerox Time-to-Market process
- **Large aerospace:** TRW CCPDS-R
- **Variety of projects:** Rational Unified Process, SPC Evolutionary Spiral Process, USC MBASE approach

A number of successful spiral approaches satisfying the spiral model invariants were presented at the workshop, often with supplementary material elsewhere. C-Bridge's RAPID approach has been used successfully to develop e-commerce applications in 12-24 weeks. Its Define, Design, Develop, and Deploy phases use the equivalent of the LCO, LCA and IOC anchor point milestones as phase gates [Leinbach, 2000]. The large spiral telecommunications applications discussed in [Bernstein, 2000] and [DeMillo, 2000] use a complementary best practice at their anchor point milestones: the AT&T/Lucent/Telcordia Architecture Review Board process [AT&T, 1993]. Xerox's Time-to-Market process uses the anchor point milestones as hardware-software synchronization points for its printer business line [Hantos, 2000].

Several successful large aerospace spiral projects were also discussed. The best documented of these is the CCPDS-R project discussed in [Royce, 1998]. Its Ada Process Model was the predecessor of the Rational Unified Process and USC MBASE approach, which have been used on a number of successful spiral projects [Jacobson et al., 1999; Boehm et al., 1998], as has the SPC Evolutionary Spiral Process [SPC, 1994].

References

(MBASE material available at <http://sunset.usc.edu/MBASE>)

- [AT&T, 1993]. "Best Current Practices: Software Architecture Validation," AT&T, Murray Hill, NJ 1993.
- [Bernstein, 2000]. L. Bernstein, "Automation of Provisioning," Proceedings, USC-SEI Spiral Experience Workshop, February 2000.
- [Boehm, 1988]. "A Spiral Model of Software Development and Enhancement," Computer, May 1988, pp. 61-72.
- [Boehm, 1989]. "Software Risk Management", IEEE Computer Society Press, 1989.
- [Boehm, 2000]. B. Boehm, "Unifying Software Engineering and Systems Engineering," IEEE Computer, March 2000, pp. 114-116.
- [Boehm et al., 1997] "Developing Multimedia Applications with the WinWin Spiral Model," Proceedings, ESEC/FSE 97, Springer Verlag, 1997.
- [Boehm et al., 1998]. "Using the Win Win Spiral Model: A Case Study," IEEE Computer, July 1998, pp. 33-44.
- [Boehm et al., 2000]. B. Boehm, M. Abi-Antoun, A.W. Brown, N. Mehta, and D. [Port, 2000]. "Guidelines for the LCO and LCA Deliverables for MBASE," USC-CSE, March 2000, http://sunset.usc.edu/classes/cs577b_2000/EP/07/MBASE_Guidelines_for_CS577v0.2.pdf
- [Boehm-Ross, 1989]. "Theory W Software Project Management: Principles and Examples" IEEE Trans. Software Engr., July 1989.
- [Boehm-Bose, 1994]. "A Collaborative Spiral Software Process Model Based on Theory W," Proceedings, ICSP 3, IEEE, Reston, Va. October 1994.
- [Boehm-Port, 1999a]. "Escaping the Software Tar Pit: Model Clashes and How to Avoid Them," ACM Software Engineering Notes, January, 1999, pp. 36-48.
- [Boehm-Port, 1999b]. "When Models Collide: Lessons from Software Systems Analysis," IEEE IT Professional, January/February 1999, pp. 49-56.
- [Carr et al., 1993]. "Taxonomy-Based Risk Identification," CMU/SEI-93-TR-06, Software Engineering Institute, 1993.
- [Charette, 1989]. Software Engineering Risk Analysis and Management, McGraw Hill, 1989.
- [Cusumano-Selby, 1995] Microsoft Secrets, Free Press, 1995

- [DeMillo, 2000]. R. DeMillo, "Continual Improvement: Spiral Software Development", Proceedings, USC-SEI Spiral Experience Workshop, February 2000.
- [Hall, 1998] Managing Risk, Addison Wesley, 1998.
- [Hantos, 2000]. P. Hantos, "From Spiral to Anchored Processes: A Wild Ride in Lifecycle Architecting", Proceedings, USC-SEI Spiral Experience Workshop, February 2000.
- [Forsberg et al., 1996]. K. Forsberg, H. Mooz, and H. Cotterman, Visualizing Project Management, Wiley, 1996.
- [Garlan et al., 1995]. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," IEEE Software, November 1995, pp. 17-26.
- [Jacobson et al., 1999]. The Unified Software Development Process, Addison-Wesley, 1999.
- [J. Thorp, 1998]. The Information Paradox, McGraw Hill, 1998.
- [Kellner et al., 1998]. "Process Guides: Effective Guidance for Process Participants," Proceedings of the 5th International Conference on the Software Process: Computer Supported Organizational Work, 1998.
- [Kitaoka, 2000]. B. Kitaoka, "Yesterday, Today & Tomorrow: Implementations of the Development Lifecycles", Proceedings, USC-SEI Spiral Experience Workshop, February 2000.
- [Kruchten, 1999]. The Rational Unified Process, Addison-Wesley, 1998.
- [Leinbach, 2000]. C. Leinbach, "E-Business and Spiral Development", Proceedings, USC-SEI Spiral Experience Workshop, February 2000.
- [Mehta, 1999]. N. Mehta, "MBASE Electronic Process Guide," USC-CSE, October 1999, http://sunset.usc.edu/classes/cs577a_99/epg
- [Royce, 1998] Software Project Management: A Unified Framework, Addison Wesley, 1998.
- [SPC, 1994]. Software Productivity Consortium, "Process Engineering with the Evolutionary Spiral Process Model," SPC-93098-CMC, version 01.00.06, Herndon, Virginia, 1994.
- [USAF, 2000]. U.S. Air Force, "Evolutionary Acquisition for C2 Systems," Air Force Instruction 63-123, 1 January 2000.

Software Lifecycle Connectors: Bridging Models across the Lifecycle

Nenad Medvidovic

Paul Gruenbacher

Alexander F. Egyed

Barry W. Boehm

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781

{neno,gruenbac,aegyed,boehm}@sunset.usc.edu

ABSTRACT

Numerous notations, methodologies, and tools exist to support software system modeling. While individual models may clarify certain system aspects, the large number and heterogeneity of models may ultimately hamper the ability of stakeholders to communicate about a system. The major reason for this is the discontinuity of information across different models. In this paper, we present an approach for dealing with that discontinuity. We propose an extensible set of “connectors” to bridge models, both within and across the activities in the software development lifecycle. While the details of these connectors are dependent upon the source and destination models, they share a number of underlying characteristics. We illustrate our approach by applying it to a large-scale system we are currently designing and implementing in collaboration with a third-party organization.

1 INTRODUCTION

The many and diverse stakeholders in a software project constantly battle the problem of conveying their concerns to other stakeholders in a manner that is understandable and that will ensure the proper treatment of those concerns. To deal with this problem, software engineering researchers and practitioners have developed a plethora of models that focus on different aspects of a software system. These models fall into five general categories: domain, success, process, product, and property models [3,6]. Numerous notations, methodologies, and tools exist to support models in each category. For example, within the last decade, the heightened interest in software architectures has resulted in several product and property models based on architecture description languages (ADLs), architectural styles, and their supporting toolsets [24,29,33].

However, this preponderance of models does not necessarily solve the problem of enabling different stakeholders to communicate. On the contrary, the increased number of models renders the ultimate goal of software engineering research—developing dependable software—even more difficult in many ways. The reason for this is the *discontinuity* of information across different models. For example, a system’s requirements will often be described using use-case scenarios and entity-relationship diagrams, while its design is captured in class, object, collaboration, and activity diagrams. The problem, then, is twofold:

1. ensuring the consistency of information across models describing the same artifact (e.g., class and collaboration diagrams), and
2. ensuring the consistency of information across models describing different artifacts (e.g., use-case scenarios and class diagrams).

In both cases, each model provides (possibly different) information in different ways, making it very difficult to establish any properties of the modeled phenomena as a whole.

In principle, this discontinuity among models can be dealt with by employing *synthesis* and *analysis*. Synthesis enables one to generate a new model (e.g., collaboration diagram) from an existing model (e.g., class diagram), while analysis provides mechanisms for ensuring the preservation of certain properties across (independently created) models. Software engineers extensively employ both kinds of techniques. For example, program compilation involves both the analysis of the syntactic and semantic correctness of one model (source code) and the synthesis of another model from it (executable image).

Synthesis and analysis techniques span a spectrum from manual to fully automated. Manual techniques tend to be error prone, while fully automated techniques are often infeasible [27]. Furthermore, in some cases one technique (e.g., analysis) is easier to perform than another (synthesis). For this reason, one typically has resort to using some combination of synthesis and analysis techniques of varying degrees of automation when ensuring inter-model consistency.

The focus of our previous work was on identifying and classifying different categories of models (domain, success, process, product, and property [3,6]) and providing support for specific models *within* each category (e.g., requirements models [2,5], architecture models [23], and design models [12]). This paper discusses a set of techniques we have developed to *bridge* the information gap created by such heterogeneous models in the context of software development.

In many ways, we view this problem as similar to the one that has recently generated much interest in the software architecture community: a software architecture can be conceptualized as a diagram consisting of “boxes,” representing components, and “lines,” representing component relationships (i.e., connectors); while we may have a more complete understanding of the components, many of the critical properties of a software system are hidden within its connectors [25,32]. Similarly, the individual models produced during a software system’s lifecycle comprise the “lifecycle architecture” boxes; the properties of these individual models are typically well understood. Much more challenging is the problem of understanding and providing the necessary support for the lines between the boxes, i.e., the model “connectors.”

The work described in this paper has focused on providing

This research is sponsored by DARPA through Rome Laboratory under contract F30602-94-C-0195 and by the Affiliates of the Center for Software Engineering

connectors for models traditionally associated with the “upstream” activities in the software lifecycle: domain analysis/requirements, architecture, and design. One reason for choosing this set of models is that, as alluded above, the “downstream” models (e.g., implementation) are typically more formal, and well-understood synthesis/analysis techniques and tools for them exist (e.g., compilation). Furthermore, the downstream models involve more homogeneous, technically savvy stakeholders (designers, developers, testers); on the other hand, the upstream models involve a broader set of more heterogeneous stakeholders (including customers, users, managers, and architects) and thus more heterogeneous models. In particular, we have devised a set of techniques for bridging

1. requirements and architecture models,
2. architecture and design models, and
3. different design models, both the same level and across levels of abstraction.

As this paper will demonstrate, each of these three categories of model connectors introduces its own issues and challenges. Furthermore, for practical reasons, our investigation has focused on a limited number of models. At the same time, we have devised a set of shared principles and techniques that are model-independent. As already discussed, we use a combination of *synthesis* and *analysis*. Furthermore, we classify the relationships among the elements of different models as *unrelated*, *complementary*, *redundant*, and *contradictory*. In each case, we use that classification to introduce *intermediate models*, allowing the interpretation of one model (e.g., requirements) in terms of another model’s vocabulary (e.g., architecture). Finally, we have provided a common integration platform for the different models’ notations and tools.

The remainder of the paper is organized as follows. Section 2 introduces the example application we will use for illustration throughout the paper. Section 3 briefly describes the requirements, architecture, and design modeling techniques (i.e., the modeling “components”) we have used as the basis of this work. Sections 4, 5, and 6 discuss the requirements-to-architecture, architecture-to-design, and inter-design model connectors, respectively. A discussion of lessons learned and conclusions round out the paper. It is important to note that our approach does not assume any particular lifecycle model (e.g., waterfall or spiral) or software development process. The sequential ordering of lifecycle activities implied by the paper’s organization (Sections 4, 5, and 6 in particular) was adopted for presentation purposes only. It is indeed possible (and often preferable [5]) to concurrently develop, relate, and refine requirements, architecture, and design models using our approach.

2 EXAMPLE APPLICATION

We use an example application to illustrate the concepts introduced in this paper. The application is motivated by the scenario we developed in the context of a U.S. Defense Advanced Research Project Agency (DARPA) project demonstration and recently refined in collaboration with a major U.S. Department of Defense (DoD) software development organization. The scenario postulates a natural disaster that

results in extensive material destruction and numerous casualties. In response to the situation, a major international humanitarian relief effort is initiated, causing several challenges from a software engineering perspective. These challenges include efficient routing and delivery of large amounts of material aid; wide distribution of participating personnel, equipment, and infrastructure; rapid response to changing circumstances in the field; using existing software for tasks for which it was not intended; and enabling the interoperability of numerous, heterogeneous systems employed by the participating countries.

We have performed a thorough requirements, architecture, and design modeling exercise to address these concerns. We have also provided a partial implementation for the resulting system (referred to as “cargo router”). This implementation is an extension of the logistics applications discussed in [23]. We are currently collaborating with the DoD organization with the goal of further extending the current implementation to address a larger portion of the system’s requirements and include several legacy components.

3 SOFTWARE MODELING

Our previous work has focused on numerous aspects of modeling software requirements, architectures, and designs. Although the focus of this paper is on model *connectors*, rather than the models themselves, a brief description of the modeling approaches is needed to provide the necessary background. Furthermore, this discussion will highlight the issues we encountered in trying to bridge the different models, ensure the consistency among them, and properly facilitate their evolution.

3.1 Requirements

The success of a software project depends highly on the involvement and interaction of important stakeholders at all stages of the software lifecycle. Throughout the lifetime of a project, and particularly during requirements engineering, stakeholder needs and goals have to be gathered, communicated, and negotiated to achieve a mutually satisfactory solution that takes into account all known objectives and constraints. We have developed the WinWin approach for collaborative requirements negotiation and successfully applied it in over 100 real-client projects [2,5,5].

WinWin provides a negotiation model that defines a set of artifacts to guide the negotiation process: stakeholder objectives and goals are expressed as *win conditions*; known constraints, problems, and conflicts among win conditions are captured as *issues*; *options* describe possible alternative solutions to overcome the issues; if a consensus is achieved among stakeholders, *agreements* are created. All WinWin artifacts are organized in a *domain taxonomy* that defines the negotiation space and typically covers application features, system properties, interfaces, and project and process artifacts [3]. A glossary of *terms* is used in WinWin to capture the domain language.

We have recently enhanced the WinWin approach and have used a state-of-the-art COTS groupware environment as its implementation substrate [17]. The result, “EasyWinWin,”

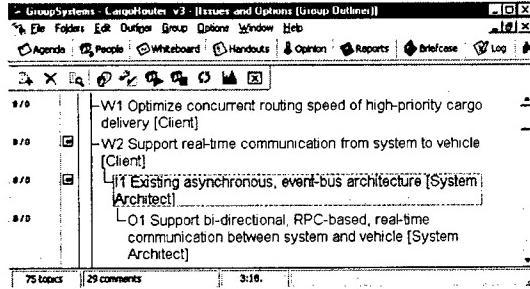


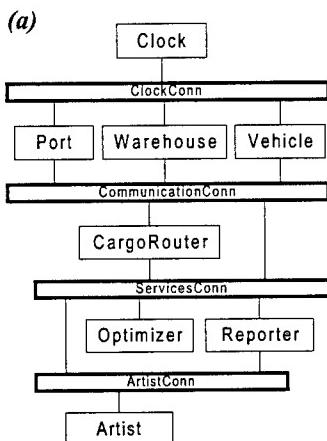
Figure 1. EasyWinWin negotiation tree.

aims at enhancing the directness, extent, and frequency of stakeholder interaction in the requirements engineering process [5,15]. EasyWinWin supports brainstorming, categorization, and prioritization of win conditions; cooperative development and refinement of domain taxonomies; shared definition of terms; and negotiations and conflict resolution following the WinWin negotiation model. A recent addition to EasyWinWin is an interface to the Rational Rose CASE tool, provided to support repository-based integration of negotiation results and to allow further analyses, reports, and traceability to other artifacts [16].

A team of stakeholders used the EasyWinWin methodology and collaboration tools to gather, negotiate, and elaborate requirements for the cargo router system. Figure 1 shows a snapshot of the EasyWinWin negotiation tool: WinWin artifacts are organized in a tree and marked with type and stakeholder/role tags.

3.2 Architectures

The architectural approach we are using in studying lifecycle connectors is the C2 architectural style [23]. We have selected C2 as a vehicle for exploring our ideas because it provides a number of useful rules for high-level system composition, demonstrated in numerous applications across several domains. C2 has been the basis of our previous work on architecture-based software modeling, analysis, generation, evolution, reuse, and heterogeneity [20,21,22,23,25].



(b) Partial cargo routing system architecture specified in C2SADEL.

```

architecture CargoRouteSystem is {
  component_types {
    component Port is extern {Port.c2;};
    component Artist is virtual {};
    ...
  }
  connector_types {
    connector RegConn is {filter no_filter;};
  }
  architectural_topology {
    component_instances {
      aPort : Port;
      Display : Artist;
    }
    connector_instances {
      ClockConn, ArtistConn : RegConn;
    }
  }
  connections {
    connector Clock Conn {
      top SimClock;
      bottom aPort;
    }
    connector ArtistConn {
      top Optim, Report, ServicesConn;
      bottom Display;
    }
  }
}
  
```

(c) Partial Port component type specified in C2SADEL. Interface and operation labels (e.g., ip_selshp) are a notational convenience. “~” denotes the value of a variable after the operation has been performed, while “#” denotes set cardinality.

```

component Port is
  subtype CargoRouteEntity (int \and beh) {
    state {
      cargo : \set Shipment; selected : Integer; ...
    }
    invariant { (cap >= 0) \and (cap <= max_cap); }
    interface {
      prov ip_selshp: Select(sel : Integer);
      req ir_clkck: ClockTick();
    }
    operations {
      prov op_selshp: {
        let num : Integer;
        pre num <= #cargo;
        post ~selected = num;
      }
      req or_clkck: {
        let time : STATE_VARIABLE;
        post ~time = time + 1;
      }
    }
    map {
      ip_selshp -> op_selshp (sel -> num);
      ir_clkck -> or_clkck();
    }
  }
}
  
```

Figure 2. (a) Architectural breakdown of the cargo routing system. (b) Partial cargo routing system architecture specified in C2SADEL. (c) Partial Port component type specified in C2SADEL. Interface and operation labels (e.g., ip_selshp) are a notational convenience. “~” denotes the value of a variable after the operation has been performed, while “#” denotes set cardinality.

An architecture in the C2 style consists of components, connectors (software buses) [25], and their configurations. Each component has two connection points, a “top” and a “bottom.” Components communicate solely by exchanging messages. The top (bottom) of a component can only be attached to the bottom (top) of one bus. It is not possible for components to be attached directly to each other: buses always have to act as intermediaries between them. However, two buses can be attached together.

The C2 architecture of a subset of the cargo routing application is shown in Figure 2a. The *Port*, *Vehicle*, and *Warehouse* component types are objects that keep track of the state of delivery ports, transportation vehicles, and warehouses, respectively. The *CargoRouter* component determines when cargo arrives at a port, keeps track of available transport vehicles at each port, and tracks the cargo during its delivery to a warehouse. The *Optimizer* tries to maximize the system’s efficiency by determining the optimal distribution of vehicles at the delivery ports, assignment of cargo to the vehicles, and routing of vehicles to the warehouses. The *Reporter* component allows runtime progress tracking of the system. Finally, *SystemClock* provides consistent time measurement to interested components, while the *Artist* component renders the user interface using a graphics toolkit (e.g., Java AWT).

C2-style architectures are modeled using an ADL—C2SADEL [23]. C2SADEL allows modeling of a component type’s state, interface, and behavior expressed in first-order logic. Furthermore, it allows modeling each connector type’s message routing policy (unicast, groupcast, multicast, broadcast), expressed in terms of message filtering (e.g., *no filtering* denotes message broadcast). Finally, C2SADEL allows the instantiation of components and connectors and the composition of those instances into a configuration. For illustration, an excerpt of a C2SADEL model of the cargo router architecture is shown in Figure 2b, while a partial specification of the *Port* component type is given in Figure 2c.

The C2 approach is supported by a family of development environments [20,23]. Of particular significance to this work is the DRADEL toolset for architecture modeling (in

C2SADEL), analysis, implementation, and evolution [23]. Similarly to EasyWinWin (recall Section 3.1), DRADEL has been recently integrated with Rational Rose [1]. As discussed in Section 5, the resulting tool suite, SAAGE, is used to support the bridging of architectural models with requirements and design models.

3.3 Design

Our work on software requirements and architectures is based on two research projects that have been on-going for the last several years. Our support for software design, on the other hand, leverages a large body of mainstream design notations and methodologies, collected into the Unified Modeling Language (UML) [8]. UML is a graphical language that provides a useful and extensible set of predefined constructs, it is semi-formally defined, and it has substantial (and growing) tool support. UML allows designers to produce several models of a software system. Each such model addresses a certain set of issues: classes and their declared attributes, operations, and relationships; the possible states and behavior of individual classes; packages of classes and their dependencies; example scenarios of system usage including kinds of users and relationships between user tasks; the behavior of the overall system in the context of a usage scenario; examples of object instances with actual attributes and relationships in the context of a scenario; examples of the actual behavior of interacting instances in the context of a scenario; and the deployment and communication of software components on distributed hosts. UML allows additional semantic constraints to be placed on its modeling elements via the Object Constraint Language (OCL), which is based on first-order predicate logic [8].

One drawback of UML is that it does not relate modeling elements across multiple UML models (i.e., diagrams). This allows for inconsistencies to be introduced among UML models with shared information. Our previous work has remedied this problem: we augment UML to provide a common, model-independent representation of a software system [12]. This representation is used as the basis of our inter-design connectors, discussed in Section 6.

4 REQUIREMENTS-TO-ARCHITECTURE CONNECTOR

We have been investigating principled ways of connecting requirements and architecture models in our effort to improve the integration between EasyWinWin and SAAGE. We faced several major challenges in developing this model connector: (1) requirements and architecture models emerge concurrently in an iterative process involving multiple stakeholders with conflicting goals, needs, expectations, and objectives; (2) the use of natural language leads to imprecision and ambiguities of requirements models; and (3) there is a semantic gap between high-level requirements and elements in a system architecture.

When viewing requirements and architecture models from a software architect's point of view, a central matter is how to systematically define a viable architecture that matches the requirements. If we look at the models from the perspective of a requirements engineer, the challenge is to effectively

employ architectural modeling and simulation to help complete the requirements and assist in detecting their inconsistencies and mismatches. Unfortunately, the large semantic gap between high-level, often ambiguous requirements artifacts, such as those introduced during WinWin negotiations, and the more specific software components and connectors, such as those modeled in C2SADEL, often does not allow one to establish meaningful links between the two models' artifacts. This section proposes an approach that remedies the problem and facilitates the bridging of the two models.

4.1 Model Connector

We have developed the CBSP (Component, Bus, System, Property) approach that provides a model connector between requirements and architectures [10,15]. CBSP artifacts refine existing WinWin artifacts (e.g., win conditions, options) to provide an intermediate model between the requirements and architectural models. Each WinWin artifact is assessed for its relevance to the system's architecture: its components, connectors (i.e., buses), overall configuration (i.e., the system itself or a particular subsystem), and their properties (e.g., reliability, performance, and cost). Based on their relevance to one or more of the CBSP dimensions, the WinWin artifacts are refined such that the architectural concerns are made explicit. For example, a win condition such as

W: The system should provide an interface to a Web browser.
can be recast into a component win condition

W_C: Netscape Navigator should be used as a component in the system.

and a connector win condition

W_B: An off-the-shelf connector should be provided to ensure interoperability with third-party components.

CBSP is a bi-directional connector; the resulting intermediate model facilitates *synthesis* of negotiation artifacts into architectural elements and enables feedback from architecture modeling and *analysis*. The CBSP dimensions include a set of architectural concerns that can be applied to systematically *classify* and *refine* negotiation artifacts and to capture architectural tradeoff *issues* and *options*. The six possible CBSP dimensions are discussed below and illustrated with examples drawn from the cargo router system negotiation.

C: artifacts that describe or involve a Component in an architecture. For example

W12 Allow customizable reports, generated on the fly.
is refined into

W12_C Report generator component.

B: artifacts that describe or imply a connector (Bus). For example

W30 The system should have interfaces to related applications (vehicle management system, staff availability).
can be refined into

W30_B Connector to staff and vehicle management system.

S: artifacts that describe System-wide features or features pertinent to multiple components and connectors. For example

W3 Capability to react to urgent cargo needs.
is refined into

W3_S The system should deploy automatic agents to monitor and react to urgent cargo needs.

CP: artifacts that describe or imply Component Properties. For example

W44 The client UI component should run on a palm-top or lap-top device.

is refined into

W44_CP The client UI component should be portable and efficient to run on palm-top as well as lap-top devices.

BP: artifacts that describe or imply connector (Bus) Properties. For example

W42 Integration of third party components should be enabled without shutting down the system.

is refined into

W42_BP Dynamic, robust connectors should be provided to enable "on the fly" component addition and removal.

SP: artifacts that describe or imply System Properties should pertain to the entire architecture. For example

W6 Operators must be notified of subsystem failures within three seconds.

is refined into

W6_SP The system should support real-time communication and awareness.

The CBSP connector enables synthesis and analysis in several ways, as discussed below.

Synthesis of architecture from requirements

Identify and classify the architectural relevance of negotiation artifacts. CBSP is applied in a voting process involving multiple experts. The experts use the six criteria described above to classify the architectural relevance of negotiation artifacts on a scale adopted from [31]: unknown, not relevant, partially, largely, fully.

Reveal incomplete and puzzling WinWin artifacts. We have found that analyzing the vote spread of the experts is a useful technique to improve the clarity and precision of requirements descriptions. Large discrepancies detected in the votes when applying CBSP indicate potentially confusing WinWin artifacts. Reframing these artifacts can help to avoid costly errors and misunderstandings. Incomplete and puzzling WinWin artifacts often represent uncertainties about an architecture's ability to satisfy a requirement and are sources of potential risk. Such artifacts can be used to prioritize risk resolution activities in a spiral solution approach [5].

Refine WinWin artifacts by splitting complex negotiation artifacts into CBSP artifacts. When bridging models expressed in a natural language with more formal

approaches, one has to handle imprecision and ambiguity. CBSP provides a principled way of viewing and refining WinWin artifacts from a software architect's perspective, thus reducing ambiguity. This approach is similar to the soft-goal interdependency graph that shows refinements of quality requirements [9].

Analysis of architecture for adherence to requirements

CBSP also supports feedback from architecture modeling to the requirements negotiation.

Capture architecture mismatches. Problems detected in architectural models and simulation can be captured as CBSP issues, such as

I12_S Three seconds system response time not possible.

Capture architecture tradeoff decisions. Architectural options and alternative solutions can be also described as CBSP elements. For example

O24_C Consider use of OTS staff management component. This capture of tradeoff decisions is similar to the ATAM technique described in [10].

4.2 Application to the Cargo Router Example

CBSP artifacts provide an intermediate model connecting the requirements and architecture models by providing comprehensible views accessible to both the requirements engineer and the software architect. Figure 3 shows an example of the use of CBSP; it depicts the relationships between partial models taken from the cargo router example. The *Negotiation Rationale View* shows a set of WinWin artifacts. The *C2 Architectural View* is a possible architecture for the cargo router example (recall Section 3.2). The CBSP model connector comprises two views: the *CBSP View*, created by classifying and refining negotiation artifacts and the *Minimal CBSP View*, created by eliminating replaced and merging related CBSP artifacts.

In the example shown in Figure 3, win condition W1 was voted as being *fully* component relevant, *largely* connector (bus) relevant, and *largely* property relevant (i.e., CP and BP). Win condition W2 and option O1 were voted as being *largely* connector (bus) relevant and *fully* connector property relevant. W3 was voted as being *largely* component relevant and issue I1 was voted as being architecturally *not relevant*. Upon further analysis, it is revealed that W1 describes multiple architectural elements. The two middle diagrams in Fig-

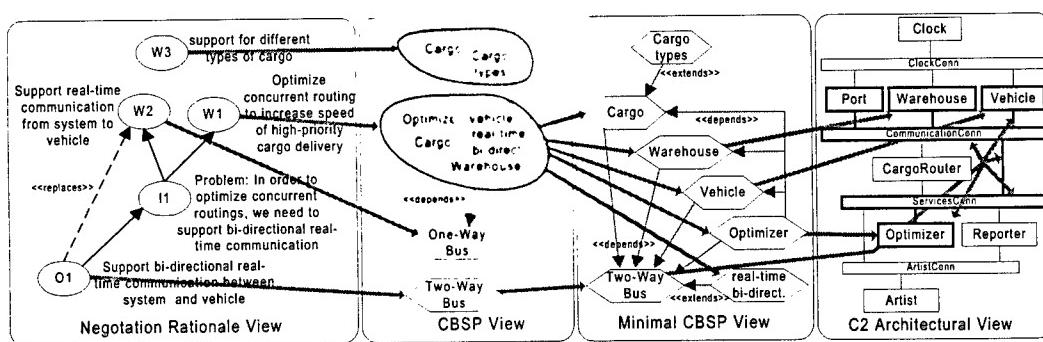


Figure 3. Synthesis of an architectural (C2) model from a requirements (WinWin) model.

ure 3 show the result of this process: W1 is divided into several components, a connector, and a connector property.

4.3 Model Connector Characteristics

Connectors between requirements and architecture models are heavily influenced by the ambiguity and imprecision of natural language. For that reason, the CBSP dimensions support a manual but guided and expert-based refinement of negotiation results. CBSP also supports analysis indirectly, by capturing architectural tradeoffs and mismatches revealed in the process of architectural modeling.

The approach highlights some of the often vague relationships between requirements negotiation artifacts and elements of a software architecture. These relationships are described in terms of our taxonomy of inter-model element relationships, introduced in Section 1.

- The voting process emphasizes architectural relevance, helping the architect to focus on the most relevant subset of the negotiation results and to ignore *unrelated* artifacts (e.g., a development schedule win condition may have no bearing on a component property win condition).
- The approach allows identifying *complementary* model elements in the requirements and architecture models by improving traceability of artifacts (e.g., a performance win condition may not be explicitly captured in an architectural model but must be taken into account).
- The approach aims at reducing *redundancy* by minimizing the CBSP view. At the same time, much of the same information will be represented in a software system's requirements and its architecture (though in different ways). CBSP highlights the relationships among such redundant modeling elements, facilitating improved stakeholder communication.
- CBSP artifacts also help to reveal *contradictory* elements and detect mismatches by exploiting the benefits of architectural modeling, analysis, and simulation in the negotiation process.

5 ARCHITECTURE-TO-DESIGN CONNECTOR

The CBSP approach presented in the preceding section suggests the key architectural elements and their properties for an application. However, in its current form, CBSP does not provide guidance for achieving an effective topology of those architectural elements: the S and SP categories of win conditions provide only hints about the characteristics of the topology. Similarly, in the course of architectural decomposition, the architect may discover that additional components and connectors are needed that have not been identified through requirements elicitation and refinement. For these reasons, CBSP must be complemented with architectural design principles.

There exists a large body of work on arriving at an effective architecture for a given problem. Architectural *styles* [33] provide rules that exploit recurring structural and interaction patterns across a class of applications and/or domains. *Domain-specific* software architectures (DSSA) and *product-line* architectures (often referred to as *application-family* architectures) [28] provide generic, reusable architectural solutions (*reference architectures*) for a class of applications

in a single domain and instantiate those solutions to arrive at a specific application architecture. Finally, a large body of architecture modeling notations—ADLs—and their supporting toolsets [24] allows developers to effectively model, analyze, implement, deploy, and evolve software systems.

5.1 Model Connector

As discussed in Section 3.2, a C2-style architecture describes a system in terms of high-level components, connectors, and their configurations. Furthermore, the style imposes certain constraints on allowed component interactions and topologies. Based on the information provided in a C2SADEL model of the architecture, the SAAGE environment is capable of generating a partial implementation of that architecture [23]. At the same time, many lower-level issues (e.g., any additional processing and data objects, specific data structures, and algorithms) that are needed to complete that implementation are not provided at the architectural level. For that reason, the “outer skeleton” of the application generated from the architectural model must be complemented with the details typically provided through lower-level design activities. In other words, a model connector is needed to bridge architectural and design models.

Software architecture researchers have studied the issue of refining an architecture into a design. An approach representative of the state-of-the-art in this area is SADL [26]. SADL incrementally transforms an architecture across levels of abstraction using a series of refinement maps, which must satisfy a correctness-preserving criterion. While powerful, this model connector can be overly stringent [13]. It sacrifices design flexibility to a notion of (absolute) correctness. The role of the human designer is virtually eliminated. Furthermore, formally proving the relative correctness of architectures at different refinement levels may prove impractical for large architectures and large numbers of levels [1]. Such an approach can be of value, however, if applied to the most critical parts of a system, and complemented by a more pragmatic model connector. We propose such a connector.

Our approach is based on enabling the synthesis of a design model from an ADL model. In particular, we choose UML as the target design language because of its wide adoption and large number of modeling features. We have conducted and in-depth study of the feasibility of mapping ADLs to UML [21,22,30]; this study involved three representative ADLs

```

Component Internal Object → Class
State Variable → Class Private Attribute
Component Invariant → Tagged Value + Class Documentation
Provided Operation → Class Operation
Required Operation → Class Documentation
Operation Pre/Post Condition → Pre/Post Condition on Class Operation
Component → <<C2-Component>> Class
Internal Object → <<C2-Component>> Class Attribute
Component Top Interface → <<Interface>> Class
Component Bottom Interface → <<Interface>> Class
Outgoing Request → <<Interface>> Class <<out>> Operation
Incoming Notification → <<Interface>> Class <<in>> Operation
Connector → <<C2-Connector>> Class
Connector Top Interface → ⋃ Bottom Interfaces of attached Components
Connector Bottom Interface → ⋃ Top Interfaces of attached Components
Architecture Configuration → Object Diagram + Component Diagram
Component/Connector Binding → Object Link (instance of an association)

```

Figure 4. Partial rule set for transforming a C2SADEL model into a UML model.

[24] (C2SADEL, Rapide [19], and Wright [2]) and two strategies for developing an ADL-to-UML model connector.

Based on this study, we have implemented a C2SADEL-to-UML connector. This connector results in an intermediate model that is, on the one hand, represented in UML, but on the other hand, it reflects the structure, details, and properties of the architectural model. The transformation from C2SADEL to this

intermediate UML model is defined by a set of rules; an excerpt of the rule set is shown in Figure 4. The most important requirement of the transformation is that the synthesized UML model be initially correct with respect to the architectural model; once the UML model is further refined, it is, of course, possible to introduce inconsistencies with the architectural model. Techniques for detecting and resolving these inconsistencies are presented in Section 6.

As already discussed, we have completed an initial integration of our DRADEL environment for architecture-based development in C2 with Rational Rose, a commercial UML modeling environment, resulting in the SAAGE toolset [1]. This integration allows automated synthesis of the intermediate UML models corresponding to C2SADEL architectures. The ultimate intent of this integration is to facilitate model connectors by providing shared representation and tool support across requirements, architecture, and design models.

5.2 Application to the Cargo Router Example

The C2 architecture of the cargo router application, discussed in Section 3.2, is mapped into several UML diagrams as indicated by the rules in Figure 4. In particular, each C2 component and connector is mapped to a specific set of UML class diagrams, representing its internal details as modeled in C2SADEL. Additionally, the overall configuration is mapped to UML component and object diagrams, showing the dependencies among the instances of the classes used to represent the components and connectors. Figure 5 shows the synthesized UML view of the cargo routing architecture. All the details of the architecture represented in C2SADEL are transferred into this intermediate model.

5.3 Model Connector Characteristics

The higher degree of formalization of architectural and design models renders this model connector easier to specify than in the case of requirements (recall Section 4). At the same time, the semantics of an ADL are typically defined explicitly, while several aspects of UML semantics remain

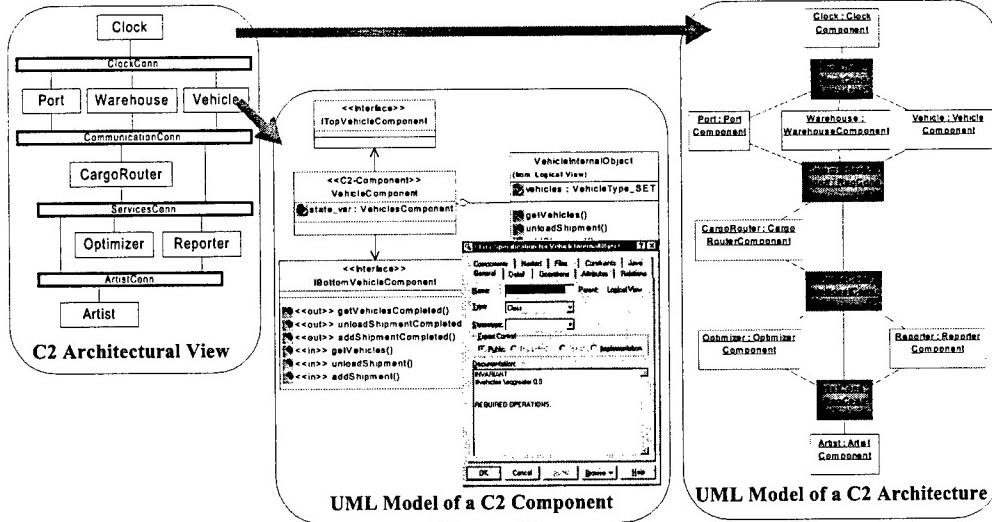


Figure 5. Synthesis of an intermediate UML model from a C2 architectural model.

implicit. We have tried to address this discrepancy and ensure the proper semantics of the intermediate UML model by placing constraints, specified in OCL, on UML modeling elements.¹ In turn, this approach was effective in revealing the relationships among architectural and design (i.e., UML) concepts:

- We made sure that the entire architectural model is transferred into the intermediate UML model. Only further refinement of the UML model will introduce elements that are potentially *unrelated* to those in the architecture.
- Certain aspects of the architectural model *complement* those in the UML model. For example, the services a C2 component requires are explicit, first-class constructs in C2SADEL and are used as the basis of architectural *analysis*. In the UML model, these services become a part of system documentation, intended as a guide to the designer.
- Given the number and diversity of UML diagrams, certain aspects of an architecture end up being mapped to multiple diagrams. For example, architectural topology is reflected in Object and Component diagrams (see Figure 4). Such *redundancy* is unavoidable when the target models have overlapping concerns. At the same time, the redundancy presents a problem in that changes in one such view must always be propagated to all other views.
- Architecture-level analysis tools are optimistically inaccurate by design since they deal with high-level, partial system models. Thus, coupling architectural models with lower-level, design models has the potential to eliminate any “false negatives” by identifying additional *inconsistencies*.

6 INTER-DESIGN CONNECTORS

Once the intermediate UML model is synthesized from the architecture, that model must be further refined to address the missing lower-level design issues, such as additional pro-

1. The OCL formulae are abstracted away in the UML stereotypes (denoted with “<>”) in Figure 5. The complete OCL specification of the C2 rules can be found in [30].

cessing and data elements, specific data structures, and algorithms. This section discusses a set of design model connectors we have developed to bridge related design models (e.g., class diagrams) at different levels of abstraction, as well as different design models (e.g., class and statechart diagrams) at the same level of abstraction. As in the preceding sections, we distinguish between synthesis methods for creating intermediate models and analysis methods for identifying inconsistencies.

6.1 Model Connectors

In order to help bridge design models, we have investigated ways of describing and identifying the causes of mismatches across UML and architectural views [12]. To this end, we have devised and applied a set of design model connectors, comprised into a view integration framework [12]. These connectors are accompanied by a set of activities and techniques for identifying inconsistencies in an automatable fashion. As in the preceding sections, our approach makes use of intermediate models.

Figure 6 depicts how synthesis and analysis can be used for bridging design models. The design model connectors can be divided into two categories: design refinement and design view connectors. Each is further discussed below.

Design refinement connectors: The upper left area of Figure 6 shows the case of bridging between higher-level and lower-level views. We already discussed one such example in Section 5, where we showed a refinement from C2 to UML (*synthesis*). Intermediate models are used to simplify that process. Similarly, intermediate models can be abstracted out of lower-level models so that they can be compared more easily with higher-level models (*analysis*).

Design view connectors: UML supports a wide range of diagrams, including sequence, collaboration, statechart, and so on. The principles of synthesis and analysis discussed above also apply to these additional types of diagrams. Our view integration framework currently encompasses eight different connectors between a variety of UML models. Each model connector yields an intermediate model (as shown in the lower right area of Figure 6) that simplifies view comparison.

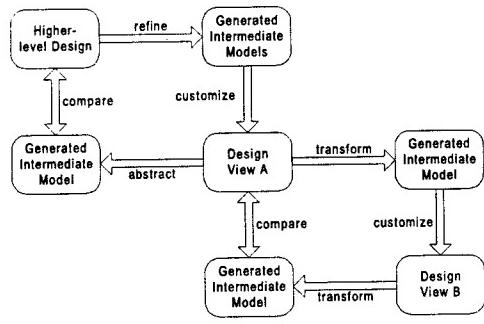


Figure 6. Consistent refinement and evolution of design models.

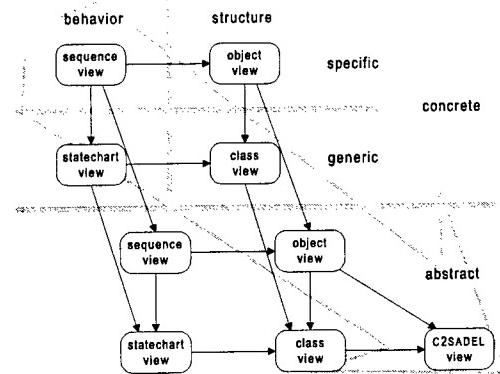


Figure 7. Design model connectors.

In our investigation of various UML views, we have identified three major transformational dimensions (see Figure 7). Views can be seen as *abstract* or *concrete*, *generic* or *specific*, and *behavioral* or *structural*. The abstract-concrete dimension was foreshadowed in Section 5, where the C2 architecture was the abstract view and the generated intermediate UML model was the concrete view. The generic-specific dimension denotes the generality of modeling information. For instance, a class diagram naturally describes a relationship between classes that must always hold, whereas an object diagram describes a specific scenario. Finally, the behavior-structure dimension takes information about behavior to infer structure. For instance, test scenarios (which are behavioral) depict interactions between objects (structural) and can thus be used to infer structure.

Manual management of design model connectors across these three dimensions is often infeasible due to the complexity of the models. Two factors contribute to the complexity: the existence of “helper classes” in a design and the overall number of classes and possible interactions. In order to control this complexity, we have developed a tool, UML/Analyzer, that uses an abstraction technique to eliminate helper classes. UML/Analyzer searches for class and object patterns and replaces them with simpler, more abstract patterns of the same type.

For instance, to identify a mismatch in the class diagram shown in Figure 8d, we need to eliminate the helper classes *availableGoods* and *aSurplus* that “obstruct” our view of the direct relationship between *aVehicle* and *aWarehouse*. In this particular example, UML/Analyzer sees an aggregation from *aVehicle* to *availableGoods*, followed by a generalization (inheritance) from *availableGoods* to *aSurplus*, which is, in turn, followed by an association from *aSurplus* to *aWarehouse* (Figure 8c). The tool then uses its abstraction rules to replace the box (class) and line (relationship) patterns.² Further applying our abstraction rules on the example, we end up finding an association relationship between *aVehicle* and *aWarehouse* (Figure 8a). This example is further discussed

2. UML/Analyzer’s rules have been created in collaboration with the Rational Software Corporation. Rational also implemented our abstraction method in a tool called Rose/Architect [11].

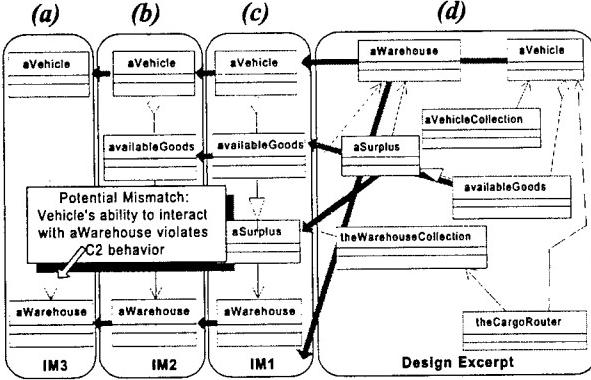


Figure 8. Series of intermediate models (from right to left) produced to identify behavioral mismatch.

below.

6.2 Application to the Cargo Router Example

In the interest of brevity, we will focus the application of design model connectors to class and object views only. We use the intermediate UML model, produced by the architecture-to-design connector discussed in Section 5, as our starting point. Figure 9 shows an excerpt of the consistency checking process in the context of the cargo router application. The figure depicts a lower-level design (right side) and its abstraction as an intermediate model (middle). The intermediate model can then be compared more easily with the original model (i.e., architecture, shown on the left side) to ensure consistency.

The components and connectors in the original, C2 model may be seen as the interfaces for compact, self-sustaining sections of the implementation. Since C2 elements are often coarse-grain, it is reasonable to assume that a collection of objects (or classes) is needed to implement each C2 element. Using UML/Analyzer, we can automatically derive an intermediate, abstracted model out of the lower-level design as discussed above. Figure 9 depicts this abstraction in the context of *Vehicle*, *Warehouse*, *CommunicationConn*, and *CargoRouter*. We can see that the association relationship between *CargoRouter* and *Vehicle* is in violation of the original architecture's structure since no corresponding link between the *CargoRouter* and *Vehicle* can be found in the C2

architecture (left side).

Another potential mismatch between the two models depicted in Figure 9 is a result of C2's rule that two components at the same level (e.g., *Vehicle* and *Warehouse*) are not allowed to directly interact. The intermediate model again helps to detect that mismatch as shown in Figure 8. The object *aVehicle* is part of *availableGoods*, which, in turn, is a child of *aSurplus*. Since *aSurplus* can only access the object *aWarehouse* (part of another component), it follows that it is possible for *Vehicle* to interact with *Warehouse*—a violation of the C2 model.

6.3 Model Connector Characteristics

The comparatively higher degree of formalization than in the case of the requirements again allows one to automatically build connectors between design models. Tools such as UML/Analyzer adopt transformation techniques to automatically *synthesize* intermediate models. These intermediate models are then used, e.g., to detect structural and behavioral inconsistencies by employing comparison (i.e., *analysis*) techniques. As it can be seen in the context of Figure 8, a series of intermediate models may be generated in response to a single transformation.

Using our taxonomy of model element relationships, we can characterize the relationships between different design models as follows:

- During design refinement, all elements of a high-level model are typically transferred into a lower-level model. However, this is not the case when bridging heterogeneous models (e.g., collaboration and statechart diagrams) at the same level of abstraction. Certain elements in such models will be *unrelated*. Another source of unrelated elements between views is further refinement of a given design model, whereby additional detail is introduced.
- An underlying principle of UML is that elements of different views *complement* each other. For example, an object diagram depicts a specific scenario, while the corresponding class diagram describes the relationship between any instances of the involved classes.
- When different concerns are expressed in UML, a certain level of *redundancy* is inevitably created. Redundant information is a prerequisite to being able to link heterogeneous views and to identify inconsistencies across them [12].
- A major purpose of design model connectors is to ensure consistency between design views at the same or different levels of abstraction by pointing out *contradictory* elements (e.g., the relation between *Vehicle* and *CargoRouter* in Figure 9). UML/Analyzer employs transformation and comparison techniques to support detection of such contradictory elements.

7 CONCLUSION

In this paper, we have discussed a set of techniques whose ultimate goal is to facilitate

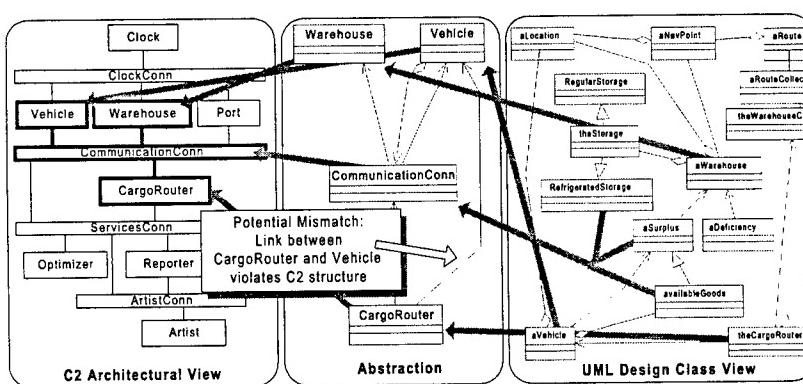


Figure 9. Use of Intermediate Model to find Structural Inconsistency

tate the consistent transformation of a system's requirements into its implementation. We believe that this is an important contribution in that our approach provides some novel solutions to a difficult problem, studied extensively by software engineering researchers. For example, the CBSP approach provides a good balance of the structure and flexibility needed to address the problem of deriving an effective architecture from a system's requirements. System quality requirements in particular tend to drive the choice of architecture [10]; at the same time, the "optimal" architecture is often a discontinuous function of the required quality level. Highly formal approaches are typically unable to adequately deal with this discontinuity, while the collaborative CBSP approach can handle it more readily. CBSP addresses the issue by involving experts in a voting process to determine the architectural relevance of negotiation artifacts and to identify incomplete and inconsistent requirements.

Another, perhaps even more important contribution of this paper lies in its identification of a set of underlying principles needed to enable a series of model transformations: all of the *model connectors* we have developed to date and discussed in this paper rely the use of *intermediate models*, the coupling of *analysis* and *synthesis* of varying degrees of *automation*, and the framework for *relating model elements* across models. While we have developed and applied these principles in the context of specific requirements, architecture, and design modeling approaches, we have taken special care to ensure their broader applicability. Thus, for example, the CBSP approach does not depend on the use of WinWin, but can instead be applied to arbitrary requirements model artifacts. Similarly, we have already applied our ADL-to-UML model connector to several ADLs [22,30].

Our work in this arena continues along several dimensions. The MBASE approach [3,6] and its support for multiple model categories is used as the conceptual integration platform for this work. We are also integrating the tool support provided by EasyWinWin, SAAGE, and UML/Analyzer to facilitate easier development and implementation of model connectors; we intend leverage all three tools' use of Rational Rose to this end. Finally, we are investigating additional model connectors that will, in particular, enable the use of multiple ADLs to enable architectural modeling of different system characteristics. We are currently studying the possibility of using the ACME architecture interchange language [14] as the intermediate model for such a model connector.

8 REFERENCES

1. M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Architecture into a Design. *UML'99*, Fort Collins, CO, October 1999.
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, July 1997.
3. Boehm B., Port D., Egyed A., Abi-Antoun M., The MBASF Life-Cycle Architecture Package, In: Donohoe P. (ed.), *Software Architecture*, Kluwer Academic Publishers, 1999.
4. B. Boehm, In. H. Identifying Quality Re-quirement Conflicts. *IEEE Software*, 13(2), March 1996.
5. B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, R. Madachy. Using the WinWin Spiral Model: A Case Study. *IEEE Computer*, 7:33-44, 1998.
6. Boehm B., and Port D., Escaping the Software Tar Pit: Model Clashes and How to Avoid Them. *ACM Software Engineering Notes*, January 1999.
7. Boehm B., Gruenbacher P., Supporting Collaborative Requirements Negotiation: The Easy-WinWin Approach. International Conference on Virtual Worlds and Simulation, San Diego 2000.
8. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
9. Chung L., Gross D., Yu E., Architectural Design to Meet Stakeholder Requirements, In: Donohoe P. (ed.), *Software Architecture*, Kluwer Academic Publishers, 1999.
10. Egyed A., Gruenbacher P., Medvidovic N. Refinement and Evolution Issues between Requirements and Architecture, Technical Report, USC-CSE, Los Angeles, CA, 2000.
11. A. Egyed and P. Kruchten. Rose/Architect: A Tool to Visualize Architecture. In *Proceedings of the Hawaii International Conference on System Sciences*, January 1999.
12. A. Egyed and N. Medvidovic. A Formal Approach to Heterogeneous Software Modeling. In Proceedings of the Conference on the Foundational Aspects of Software Engineering, 2000.
13. D. Garlan. Style-Based Refinement for Software Architecture. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pp. 72-75, San Francisco, CA, October 1996.
14. D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.
15. P. Gruenbacher. Collaborative Requirements Negotiation with Easy-WinWin, Technical Report, USC-CSE, 2000.
16. Gruenbacher P., Egyed A., Medvidovic N. Dimensions of Concerns in Requirements Negotiation and Architecture Modeling, Technical Report, USC-CSE, Los Angeles, CA, 2000.
17. GroupSystems.com. <http://www.groupsystems.com/>
18. Kazman R., Barbacci M., Klein M., Carriere, S.J., Woods S.G., Experience with Performing Architecture Tradeoff Analysis, ICSE 99.
19. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, September 1995.
20. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. *ICSE'97*, Boston, MA, May 1997.
21. N. Medvidovic and D.S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. *First IFIP Working Conference on Software Architecture (WICSA1)*, pp. 161-182, San Antonio, TX, February 1999.
22. N. Medvidovic, D. S. Rosenblum, J. E. Robbins, and D. F. Redmiles. Modeling Software Architectures in the Unified Modeling Language. Submitted for publication, 1999.
23. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. *ICSE'99*.
24. N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Accepted for publication in *IEEE Transactions on Software Engineering*, January 2000.
25. Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. *ICSE 2000*.
26. M. Moriconi, X. Qian, and R. A. Riemschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 356-372, April 1995.
27. H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, vol. 15, no. 3, pp. 199-236, September 1983.
28. D. E. Perry. Generic Descriptions for Product Line Architectures. In *Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families (ARES II)*, Las Palmas de Gran Canaria, Spain, February 1998.
29. D. E. Perry and A. L. Wolf. "Foundations for the Study of Software Architectures." *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pages 40-52, October 1992.
30. J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *ICSE'98*, Kyoto, Japan, 1998.
31. Rout, T. P. SPICE: A framework for software process assessment. *Journal Software Process Improvement and Practice*, Pilot Issue, John Wiley & Sons, 1995.
32. M. Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Proceedings of the Workshop on Studies of Software Design*, 1993.
33. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

Using the WinWin Spiral Model: A Case Study

Fifteen teams used the WinWin spiral model to prototype, plan, specify, and build multimedia applications for USC's Integrated Library System. The authors report lessons learned from this case study and how they extended the model's utility and cost-effectiveness in a second round of projects.

Barry Boehm

*Alexander
Egyed*

Julie Kwan

Dan Port

Archita Shah

University of
Southern
California

Ray Madachy

Litton Data
Systems and
University of
Southern
California

At the 1996 and 1997 International Conferences on Software Engineering, three of the six keynote addresses identified negotiation techniques as the most critical success factor in improving the outcome of software projects. At the USC Center for Software Engineering, we have been developing a negotiation-based approach to software system requirements engineering, architecture, development, and management. Our approach has three primary elements:

- *Theory W*, a management theory and approach, which says that making winners of the system's key stakeholders is a necessary and sufficient condition for project success.¹
- *The WinWin spiral model*, which extends the spiral software development model by adding Theory W activities to the front of each cycle. The sidebar "Elements of the WinWin Spiral Model" describes these extensions and their goals in more detail.
- *WinWin*, a groupware tool that makes it easier for distributed stakeholders to negotiate mutually satisfactory (win-win) system specifications.²

In this article, we describe an experimental validation of this approach, focusing on the application of the WinWin spiral model. The case study involved extending USC's Integrated Library System to access multimedia archives, including films, maps, and videos. The Integrated Library System is a Unix-based, text-oriented, client-server COTS system designed to manage the acquisition, cataloging, public access, and circulation of library material. The study's specific goal was to evaluate the feasibility of using the WinWin spiral model to build applications written by USC graduate student teams. The students developed the applications in concert with USC library clients, who had identified many USC multimedia archives that seemed worthy of transformation into digitized, user-interactive archive management services.



The study showed that the WinWin spiral model is a good match for multimedia applications and is likely to be useful for other applications with similar characteristics—rapidly moving technology, many candidate approaches, little user or developer experience with similar systems, and the need for rapid completion. The study results show that the model has three main strengths.

- *Flexibility*. The model let the teams adapt to accompanying risks and uncertainties, such as a rapid project schedule and changing team composition.
- *Discipline*. The modeling framework was sufficiently formal to maintain focus on achieving three main, or "anchor-point," milestones: the life-cycle objectives, the life-cycle architecture, and the initial operational capability. (Table A in the sidebar describes these milestones.)
- *Trust enhancement*. The model provided a means for growing trust among the project stakeholders, enabling them to evolve from adversarial, contract-oriented system development approaches

Elements of the WinWin Spiral Model

The original spiral model¹ uses a cyclic approach to develop increasingly detailed elaborations of a software system's definition, culminating in incremental releases of the system's operational capability. Each cycle involves four main activities:

- Elaborate the system or subsystem's product and process objectives, constraints, and alternatives.
- Evaluate the alternatives with respect to the objectives and constraints. Identify and resolve major sources of product and process risk.
- Elaborate the definition of the product and process.
- Plan the next cycle, and update the life-cycle plan, including partition of the system into subsystems to be addressed in parallel cycles. This can include a plan to terminate the project if it is too risky or infeasible. Secure the management's commitment to proceed as planned.

Since its creation, the spiral model has been extensively elaborated² and successfully applied in numerous projects.^{3,4} However, some common difficulties led USC-CSE and its affiliate organizations to extend the model to the WinWin spiral model described in the main text.

Negotiation front end

One difficulty was determining where the elaborated objectives, constraints, and alternatives come from. The WinWin spiral model resolves this by adding three activities to the front of each spiral cycle, as Figure A shows.⁵

- Identify the system or subsystem's key stakeholders.
- Identify the stakeholders' win conditions for the system or subsystem.
- Negotiate win-win reconciliations of the stakeholders' win conditions.

We have found in experiments with a bootstrap version of the WinWin groupware tool that these steps do indeed pro-

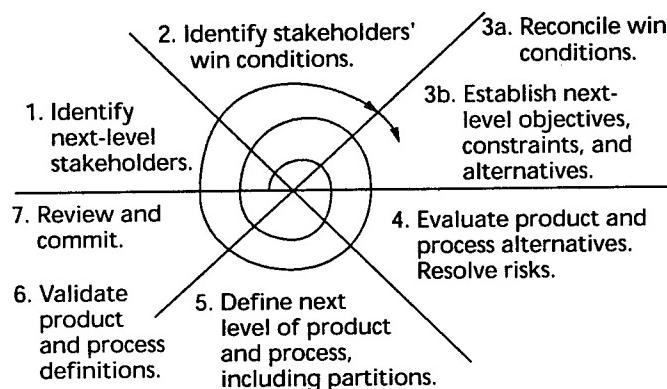


Figure A. How the WinWin spiral model differs from the original spiral model. The new model adds front-end activities (blue) that show where objectives, constraints, and alternatives come from. This lets users more clearly identify the rationale involved in negotiating win conditions for the product.

toward methods that were mutually supportive and cooperative.

From lessons learned during the case study, we identified several possible enhancements, some of which we made. We then used the enhanced model on 16 projects in the following year. The second-year projects overcame many of the weaknesses in the first-year projects. We are incorporating improvements identified by student critiques and are planning third-year projects. Industry is also looking at the WinWin spiral model. Companies such as Rational Inc. have already adopted several elements of the WinWin spiral model as part of their project management and product life-cycle processes.

duce the key product and process objectives, constraints, and alternatives for the next version.⁶ The model includes a stakeholder WinWin negotiation approach that is similar to other team approaches for software and system definition such as gIBIS, Viewpoints, Participatory Design, and Joint Application Design. However, unlike these and other approaches, we use the stakeholder win-win relationship as the success criterion and organizing principle for software and system definition. Our negotiation guidelines are based on the Harvard Negotiation Project's techniques.⁷

Process anchor points

Another difficulty in applying the spiral model across an organization's various projects was that the organization has no common reference points for organizing its management procedures, cost and schedule estimates, and so on. This is because the cycles are risk driven, and each project has different risks. In attempting to work out this difficulty with USC-CSE's industry and government affiliates using our Cocomo II cost model, we found a set of three process milestones, or *anchor points*,⁸ which we could relate to both the completion of spiral cycles and to the organization's major decision milestones.

The *life-cycle objectives* (LCO) and the *life-cycle architecture* (LCA) milestones ratify the stakeholders' commitment to a feasible and consistent package of the six key milestone elements shown in Table A for the LCO anchor point.

The LCO version focuses on establishing a sound business case for the package. It need only show that there is at least one feasible architecture.

The LCA version commits to a single choice of architecture and elaborates it to the point of covering all major sources of risk in the system's life cycle.⁸ The LCA is the most

MODEL APPLICATION

We applied the WinWin spiral model in four cycles:

- *Cycle 0.* Determine the feasibility of an appropriate family of multimedia applications.
- *Cycle 1.* Develop life-cycle objectives (LCO milestone), prototypes, plans, and specifications for individual applications and verify the existence of at least one feasible architecture for each application.
- *Cycle 2.* Establish a specific, detailed life-cycle architecture (LCA milestone), verify its feasibility, and determine that there are no major risks in satisfying the plans and specifications.
- *Cycle 3.* Achieve a workable initial operational capability (IOC milestone) for each project

critical milestone in the software system's life cycle. As an analogy, it is similar to the commitment you make in getting married (just as LCO is like getting engaged and IOC like having your first child).

The *initial operational capability*, or IOC, anchor point has three key elements:⁸

- Software preparation, including both operational and support software with appropriate commentary and documentation; data preparation or conversion; the necessary licenses and rights for COTS and reused software, and appropriate operational readiness testing.
- Site preparation, including facilities, equipment, supplies, and COTS vendor support arrangements.
- User, operator, and maintainer preparation, including selection, team building, training, and other qualifications for familiarization use, operations, or maintenance.

We found that the LCO and LCA milestones are highly compatible with the successful architecture review board practice pioneered by AT&T and Lucent Technologies.⁹ We used board sessions about 20 percent of the time in the first-year projects and 100 percent of the time in the second-year projects, with much better results.

References

1. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, pp. 61-72.
2. "Process Engineering with the Evolutionary Spiral Process Model: Version 01.00.06," Tech. Report SPC-93098-CMC, Software Productivity Consortium, Herndon, Va., 1994.
3. W. E. Royce, "TRW's Ada Process Model for Incremental Development of Large Software Systems," *Proc. 12th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 2-11.
4. T. Frazier and J. Bailey, "The Costs and Benefits of Domain-Oriented Software Reuse: Evidence from the STARS Demonstration Projects," IDA Paper P-3191, Institute for Defense Analyses, Alexandria, Va., 1996.
5. B. Boehm and P. Bose, "A Collaborative Spiral Software Process Model Based on Theory W," *Proc. Int'l Conf. Software Process*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 59-68.
6. B. Boehm et al., "Software Requirements as Negotiated Win Conditions," *Proc. Int'l Conf. Requirements Eng.*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 74-83.
7. R. Fisher and W. Ury, *Getting to Yes*, Penguin Books, New York, 1981.
8. B. Boehm, "Anchoring the Software Process," *IEEE Software*, July 1996, pp. 73-82.
9. "Best Current Practices: Software Architecture Validation," AT&T, Murray Hill, N.J., 1993.

Table A. Contents of the LCO milestone.

Milestone element					
Definition of operational concept	Definition of system requirements	Definition of system and software architecture	Definition of life-cycle plan	Feasibility rationale	System prototype(s)
Top-level system objectives and scope Environment parameters and assumptions Evolution parameters Operational concept Operations and maintenance scenarios and parameters Organizational life-cycle responsibilities (stakeholders)	Top-level functions, interfaces, quality attribute levels, including: Growth vectors Priorities Stakeholders' concurrence on essentials	Top-level definition of at least one feasible architecture Physical and logical elements and relationships Choices of COTS and reusable software elements Identification of infeasible architecture options	Identification of life-cycle stakeholders Users, customers, developers, maintainers, interoperators, general public, others Identification of life-cycle process model Top-level stages, increments Top-level WWWWWWHH (Why, What, When, Who, Where, How, How Much?) by stage	Assurance of consistency among elements above Via analysis, measurement, prototyping, simulation, etc. Business case analysis for requirements, feasible architectures	Exercise key usage scenarios Resolve critical risks

including system preparation, training, use, and evolution support for users, administrators, and maintainers.

We used Theory W in all the cycles, but we used the WinWin groupware tool in Cycle 1 only, because this is where it currently works best.

Cycle 0: Application family

From 1993 to 1996, the USC Center for Software Engineering (CSE) experimented with teaching the WinWin spiral model in its master's software engineering course, taught by Barry Boehm. The experiments involved using hypothetical applications, one of which was an advanced library application. Some

of the library staff became interested in having the CSE students develop useful USC library applications.

The CSE in turn had been looking for a source of new applications on which to test the WinWin spiral model. So in the summer of 1996 we met with some of the library staff to explore respective win conditions and to determine if we could identify a feasible set of life-cycle objectives for a family of USC library applications. Table 1 summarizes the win conditions for the three primary stakeholders: the library information technology community; the library operations community (including users); and the CSE.

As the table indicates, the library information technology community was energized by their dean's vision to accelerate the libraries' transition to digital

Table 1. Win conditions for the three primary stakeholders in the case study.

Library Information Technology Community	Library Operations Community	Center for Software Engineering
Accelerated transition to digital library capabilities; vision of Dean of the University Libraries	Continuity of service	Similarity of projects (for fairness, project management)
Evaluation of emerging multimedia archiving and access tools	No disruption of ongoing transition to SIRSI-based Library Information System	Reasonable match to the WinWin spiral model
Empowering library multimedia users	Career growth opportunities for system administrators	15-20 projects at 5-6 students per team
Enhancement of library staff capabilities in digital library services	No disruption of USC network operations and services	Achieve a meaningful life-cycle architecture in one semester
Leveraging of limited budget for advanced applications	More efficient operations via technology	Achieve a meaningful initial operational capability in two semesters
		Adequate network, computer, and infrastructure resources

capabilities. However, there was little budget for evaluating emerging multimedia technology and developing exploratory applications.

The library operations community and its users were already undergoing a complex transition to the new Integrated Library System. They were continually looking for new technology to enhance their operations. But they were also highly sensitive to the risks of disrupting services, and they had limited resources to experiment in new areas.

The biggest risk identified in Cycle 0 was the risk of having too many different applications and losing control of the project. Achieving a meaningful IOC in two semesters, a win condition for CSE, meant following a rapid project schedule. Because the students would be unfamiliar with both one another and with their library applications and clients, they could easily go off in all directions. We resolved this risk by focusing on a single application area—library multimedia archive services—and by developing a common domain model and set of product guidelines for all teams to follow.

Cycle 1: Application life-cycle objectives

Figure 1 shows the project guidelines we negotiated with the library staff during Cycle 0 and provided to the CS students on the first day of class. The guidelines allowed 2.5 weeks for the students to organize themselves into teams and 11.5 weeks to complete the life-cycle objective and life-cycle architecture milestones.

We also gave each project some guidelines for developing five documents (in the artifacts list under “Project Objectives” in Figure 1), including recommended page budgets. Each team had to develop two versions, one for the LCO milestone and an elaboration for the LCA milestone. To ensure that everyone used a common development process, we gave the teams a sample multimedia archive prototype and a domain model for a typical information archive extension. The domain model, in Figure 2, identifies the key stakeholders involved in such systems and key concepts like the system boundary, the boundary between the system being developed and its environment.

The project guidelines and domain model were key

to the teams’ rapid progress because they provided a common development perspective. The course lectures followed the WinWin spiral model. We began with overviews of the project artifacts (in Figure 1 under “Project objectives”) and how they fit together. We continued with a discussion of the key planning and organizing guidelines. In later lectures, we provided more detail on the project artifacts and had guest lectures on library operations and the SIRSI system and on technological aspects such as user interface design and multimedia system architecture.

We focused each team during Cycle 1 by having them use the WinWin groupware tool for requirements negotiation.² “WinWin user negotiations” in Figure 1 identifies the four key forms in the WinWin negotiation model (win conditions, issues, options, and agreements), and their relationships. It also summarizes the stakeholder roles (developer, customer, and user) to be played by the team members. To minimize disruption to library operations, we had the operational concept and requirements team members enter the user artifacts, rather than the librarians themselves.

Figure 3a shows the final list of applications and the teams required to develop them. We ended up with 12 applications and 15 development teams, comprising both on- and off-campus students. We let the project teams select their own members to mitigate the risk of forming teams with incompatible people and philosophies. Most teams had six people.

Figure 3b shows two problem statements prepared by the library clients. These statements are much less detailed than a typical requirements set in an industrial application. The team had to go from short statements like this to a consistent set of prototypes, plans, and specifications (typically 200 pages) in 11 weeks.

To help them organize and navigate the WinWin artifacts and control the associated terminology, we gave each team a domain taxonomy and guidelines for relating the taxonomy elements to elements of the requirements specification. Figure 4 shows part of the taxonomy and guidelines.

Figure 5 shows the look and feel of the WinWin tool. In the lower right is a win condition form entered

Project objectives

Create the artifacts necessary to establish a successful life-cycle architecture and plan for adding a multimedia access capability to the USC Library Information System. These artifacts are

1. An operational concept definition
2. A system requirements definition
3. A system and software architecture definition
4. A prototype of key system features
5. A life-cycle plan
6. A feasibility rationale, assuring the consistency and feasibility of items 1-5.

Team structure

Each of the six team members will be responsible for developing the LCO and LCA versions of one of the six project artifacts. In addition, the team member responsible for the feasibility rationale will serve as project manager with the following primary responsibilities:

- Ensure consistency among the team members' artifacts (and document this in the rationale).
- Lead the team's development of plans for achieving the project results and ensure that project performance tracks the plans.

Project approach

Each team will develop the project artifacts concurrently, using the WinWin spiral approach defined in the article "Anchoring the Software Process." There will be two critical project milestones: the life-cycle objectives (LCO) and life-cycle architecture (LCA).

The LCA package should be sufficiently complete to support development of an initial operational capability (IOC) version of the planned multimedia access capability by a CS577b student team during the spring 1997 semester. The life-cycle plan should establish the appropriate size and structure of the development team.

WinWin user negotiations

Each team will work with a representative of a community of potential users of the multimedia capability (art, cinema, engineering, business, etc.) to determine that community's most significant multimedia access needs and to reconcile these needs with a feasible implementation architecture and plan. The teams will accomplish this reconciliation by using the USC WinWin groupware support system for requirements negotiation. This system provides WinWin forms for stakeholders to express their win conditions for the system, to define issues dealing with conflicts among win conditions, to support options for resolving the issues, and to consummate agreements to adopt mutually satisfactory (win-win) options.

There will be three stakeholder roles:

- *Developer.* The architecture and prototype team members will represent developer concerns, such as the use of familiar packages, stability of requirements, availability of support tools, and technically challenging approaches.
- *Customer.* The plan and rationale team members will represent customer concerns, such as the need to develop an IOC in one semester, limited budgets for support tools, and low-risk technical approaches.
- *User.* The operational concept and requirements team members will work with their designated user-community representative to represent user concerns, such as particular multimedia access features, fast response time, friendly user interface, high reliability, and flexibility of requirements.

Major milestones

September 16	All teams formed
October 14	WinWin negotiation results
October 21, 23	LCO reviews
October 28	LCO package due
November 4	Feedback on LCO package
December 6	LCA package due, individual critique due

Individual project critique

The project critique is to be done by each individual student. It should be about 3-5 pages, and should answer the question, "If we were to do the project over again, how would we do it better—and how does that relate to the software engineering principles in the course?"

Figure 1. Guidelines given to the 15 teams on how to conduct their respective multimedia archive projects. Because each project team received the same guidelines, the teams were able to progress rapidly in specifying and building the applications.

by one of the team members on the Hancock Library photo archive project, expressing the need to accommodate future upgrades, such as different image formats. The graph at the top shows how the win condition "swong-WINC-5" is linked to other WinWin forms such as issues, options, and agreements. The taxonomy helps categorize these forms into common concerns, such as those that affect user controls.

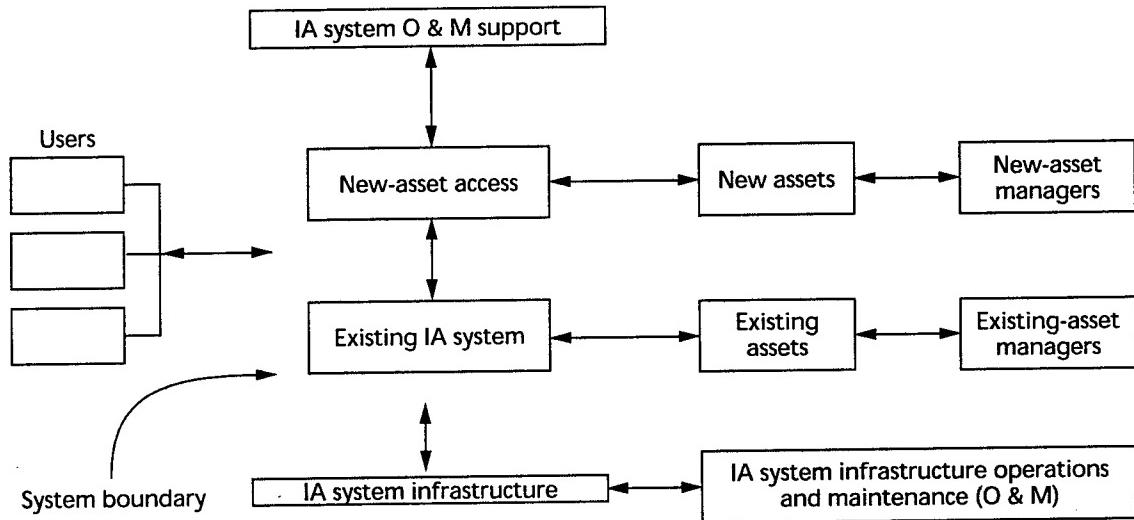
The WinWin negotiation period took longer than we expected for several reasons. We underestimated

the amount of WinWin training needed and the complexities of supporting 15 simultaneous negotiations, some with mixes of on- and off-campus negotiators. As a result, we moved the deadline for completing the WinWin negotiations and the LCO packages back a week. Fortunately, the LCO packages were good enough to let us make up that time in the next cycle.

Under the revised schedule, all 15 teams delivered their LCO packages on time. The degree of completeness was generally appropriate, but components

System block diagram

This diagram shows the usual block diagram for extensions providing access to new information archive assets from an existing information archive (IA) system:



The system boundary focuses on the automated applications portion of the operation and defines such external entities as users, operators, maintainers, assets, and infrastructure (campus networks, etc.) as part of the system environment. The diagram abstracts out such additional needed capabilities as asset catalogs and direct user access to O&M support and asset managers. Some stakeholder roles and responsibilities include:

- **Asset managers.** Furnish and update asset content and catalog descriptors. Ensure access to assets. Provide accessibility status information. Ensure asset-base recoverability. Support problem analysis, explanation, training, instrumentation, operations analysis.
- **Operators.** Maintain high level of system performance and availability. Accommodate asset and services growth and change. Protect stakeholder privacy and intellectual property rights. Support problem analysis, explanation, training, instrumentation, operations analysis.
- **Users.** Obtain training. Access system. Query and browse assets. Import and operate on assets. Establish, populate, update, and access asset-related user files. Comply with system policies. Provide feedback on use.
- **Application software maintainer.** Perform corrective, adaptive, and perfective (tuning, restructuring) maintenance on software. Analyze and support prioritization of proposed changes. Plan, design, develop, and verify selected changes. Support problem analysis, explanation, training, instrumentation, and operations analysis.
- **Service providers (network, database, or facilities management services).** Roles and responsibilities similar to asset managers.

Figure 2. Domain model for extending an information archive system. The domain model and the guidelines in Figure 1 helped give the 15 teams a unified perspective of project development.

often had serious inconsistencies in assumptions, relationships, and terminology. Most teams had planned time for members to review each others' artifacts, but most individual members ended up using that time to finish their artifacts. Some concepts—such as the nature of the system boundary, organizational relationships, and the primary goal of the life-cycle plan—caused problems for students without industrial experience. We covered these concepts in more depth in subsequent course lectures.

Cycle 2: Application life-cycle architectures

In Cycle 2, the teams chose a specific life-cycle architecture for their applications and elaborated the content of their LCO artifacts to the level of detail required for the LCA milestone. This included responding to the instructors' comments on their LCO packages. The most frequent problems were inconsistencies among the artifacts, failure to specify quality attributes, a general misunderstanding about the

application's scope (the system boundary in Figure 2a) and the inability to recognize that the plan was to focus on the development activities in Cycle 3.

Because of delays and changes in prototyping equipment, the teams developed their prototypes in Cycle 2. This had the unfortunate effect of destabilizing some of the WinWin agreements and product requirements. Once the library clients saw the prototypes, they wanted to change the requirements (the IKIWISI—I'll know it when I see it—syndrome). In the following year, we had the teams do the initial prototyping and WinWin negotiations concurrently.

On the positive side, the prototypes generally expanded the librarians' perceptions of what the teams could produce. The librarian who proposed the Edgar corporate data problem was amazed with the end product, which built on the seemingly simple text-formatting problem and delivered a one-stop Java site that synthesized several kinds of business information. She commented in her evaluation memo:

Team	Application
1.	Stereoscopic slides
2.	Latin American pamphlets
3.5.	Edgar corporate data
4.	Medieval manuscripts
6,10.	Hancock Library photo archive
7.	Interactive TV courseware delivery
8,11.	Technical reports archives
9.	Student film archive
12.	Student access to digital maps
13.	Los Angeles regional history photos
14.	Korean-American museum
15.	Urban planning documents

(a)

Medieval manuscripts

I am interested in how to scan medieval manuscripts so that a researcher could both read the content and study the scribe's hand, special markings, and so on. A related issue is how to transmit such images.

Edgar corporate data

Increasingly the government is using the WWW as a tool for dissemination of information. Two much-used sites are the Edgar database of corporate information (<http://www.sec.gov/edgarhp.htm>) and the Bureau of the Census (<http://www.census.gov>). Part of the problem is that some information (particularly that at the Edgar site) is available only as ASCII files. For textual information, the formatting of statistical tables is often lost in downloading, e-mailing, or transferring to statistical programs. While this information is useful for the typical library researcher, it is often too much trouble to put it in a usable format.

(b)

Figure 3. (a) Proposed multimedia applications and (b) two problem statements prepared by the library clients. The numbers in (a) designate the teams that designed the application. Because we had 15 teams and 12 applications, some library clients agreed to work with two teams. From statements like those in (b), the teams had to generate detailed specifications in 11 weeks.

1. Operational Modes

- 1.1 Classes of Service (research, education, general public)
- 1.2 Training
- 1.3 Graceful Degradation and Recovery

2. Capabilities

- 2.1 Media Handled
 - 2.1.1 Static (text, images, graphics, etc.)
 - 2.1.2 Dynamic (audio, video, animation, etc.)
- 2.2 Media Operations
 - 2.2.1 Query, Browse
 - 2.2.2 Access
 - 2.2.3 Text Operations (find, reformat, etc.)
 - 2.2.4 Image Operations (zoom in/out, translate/rotate, etc.)
 - 2.2.5 Audio Operations (volume, balance, forward/reverse, etc.)
 - 2.2.6 Video/Animation Operations (speedup/slowdown, forward/reverse, etc.)
 - 2.2.7 Adaptation (cut, copy, paste, superimpose, etc.)
 - 2.2.8 File Operations (save, recall, print, record, etc.)
 - 2.2.9 User Controls
- 2.3 Help
- 2.4 Administration
 - 2.4.1 User Account Management
 - 2.4.2 Use Monitoring and Analysis

3. Interfaces

- 3.1 Infrastructure (SIRSI, UCS, etc.)
- 3.2 Media Providers
- 3.3 Operators

4. Quality Attributes

:

The taxonomy serves as a requirements checklist and navigation aid:

The taxonomy elements map onto the requirements description table of contents in the course notes. Every WinWin artifact should point to at least one taxonomy element (modify elements if appropriate). Every taxonomy element should be considered as a source of potential stakeholder win conditions and agreements.

"[The team] obviously looked beyond the parameters of the problem and researched the type of information need the set of data meets. My interactions with the team were minimal, not because of any difficulty, but because as a group they had a synergy and grasped the concepts presented to them. The solution the team came up with was innovative, with the potential to be applied to other, similar problems."

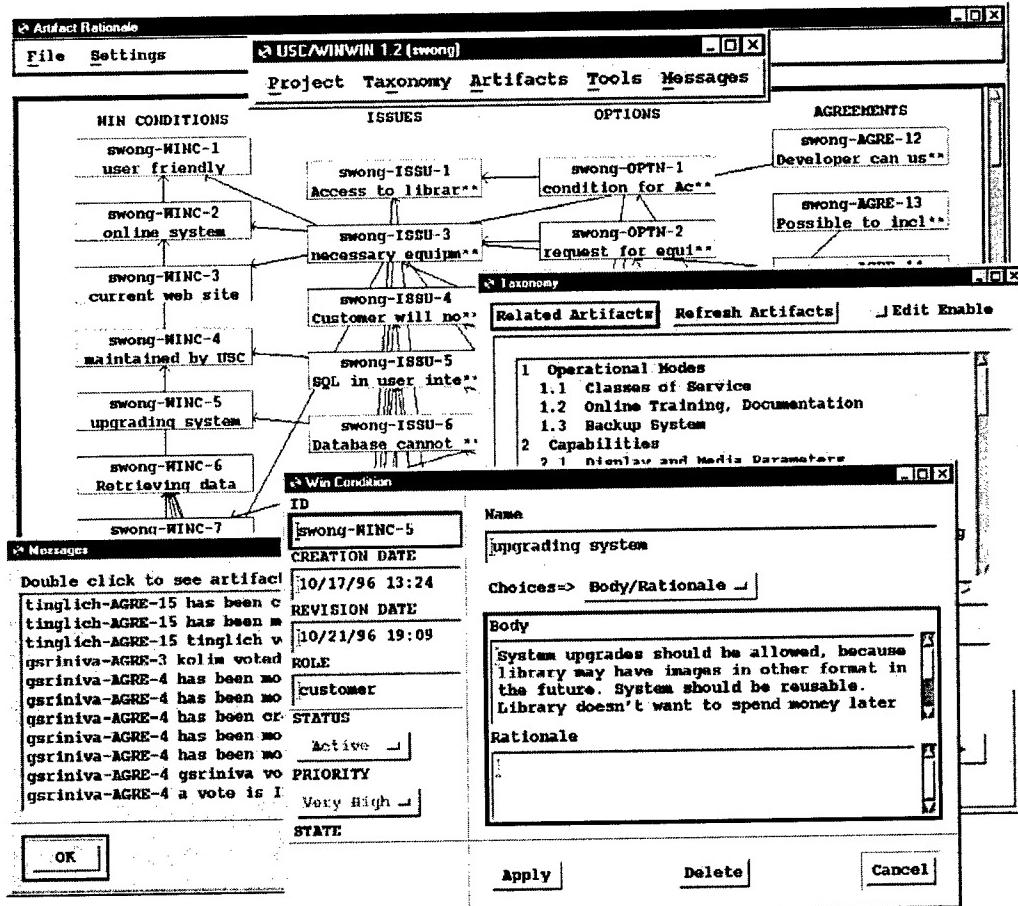
Other library clients were also very satisfied with the

value added relative to their time invested.

The teams were able to surmount several challenges characteristic of real-world projects. For example, they could not use the Integrated Library System's test server for their prototypes because it was needed in transitioning from the old system to the new Integrated Library System. There were also delays in arranging for a suitable alternative Web server. At times librarians could not provide input on critical decisions, which led to extra rework. Inevitable per-

Figure 4. Part of the domain taxonomy and use guidelines given to each project team. The taxonomy specializes the WinWin tool to the stakeholders' domain, and serves as a checklist for completing the negotiation process. It also helped the teams organize the WinWin forms and relate them to the requirements specification.

Figure 5. Sample screens from the WinWin groupware tool. The artifact rationale window (upper left) lets users immediately see links among the project stakeholders' win conditions, issues, options, and negotiated agreements. The screen in the lower right expands one of these forms, a stakeholder's win condition. The current negotiation outline is the taxonomy window (middle right) and the latest changes are listed in the message window (bottom left).



sonnel conflicts arose within the 15 teams. However, we were able to minimize conflicts within a team, in large part because the teams were self-selected.

The WinWin spiral model's mix of flexibility and discipline let the project teams adapt to these challenges while staying on schedule. In particular, the use of risk management and a continuously evolving top *n* risk list³ helped the teams focus their effort on the most critical success factors for their projects.

Another difficulty was maintaining consistency across multiple product views. The guidelines we gave the students were from the course textbook,⁴ evolving commercial standards like J-STD-016-1995, and object-oriented methods, particularly the Booch method and Object Modeling Technology. The views included system block diagrams, requirements templates, use scenarios, physical architecture diagrams, class hierarchies, object interaction diagrams, dataflow diagrams, state-transition diagrams, data descriptions, and requirements traceability relations. Each had its value, but the overall set was both an overkill and weakly supported by integrated tools. In the following year, we used a more concise and integrated set of views based on the Rational Unified Modeling Language and tool set.⁵

Cycle 3: Initial operational capability

A major challenge for Cycle 3 was that many students involved in Cycles 1 and 2 during the fall 1996 Software Engineering I course, which was a core course

for an MS in computer science, did not take Software Engineering II in spring 1997 because it was not a core course. Thus, we were able to continue only six projects during Cycle 3, involving 28 students and eight applications. The projects that continued were driven by the project experience of the students who reenrolled, rather than by the priorities of the librarians.

Only one team retained most of its LCO/LCA participants for Cycle 3. The other teams had to work with a mix of participants with varying project backgrounds. This was particularly challenging when we had to integrate teams that had produced different LCA artifacts for the same application. In two cases, the instructors had to persuade students to join different teams because they continued to fight about whose architecture was better. Other conflicts developed within teams in which some members had extensive LCA experience on the application and others had none. In one case, experienced members exploited those less experienced; in another case, the reverse happened.

Other challenges included changes in course instructor, process model (spiral to risk-driven waterfall), and documentation approach (*laissez-faire* to put-everything-on-the-Web). There were also infrastructure surprises: the Integrated Library System's server and search engine, which we expected to be available for Cycle 3, were not.

Risk management. Despite these obstacles, each project successfully delivered its IOC package—code, life-

The librarians said that working with Theory W and the WinWin philosophy made it easy for them to "think big" about their projects.

cycle documentation, and demonstrations—on time. We believe a major reason was our strong emphasis on risk management, which enabled teams to depart from a pure waterfall approach to resolve whatever critical risk items surfaced. We had each team form a top-*n* risk list, which helped them characterize each cycle and gave everyone a flavor of what to expect.

The risk list helped the team prioritize risks by assessing risk exposure (probability of loss times magnitude of loss). Each week, the team reassessed the risk to see if its priority had changed or to determine how much progress had been made in resolving it. A key strategy was design to schedule, in which a team identified a feasible core capability and optional features to be implemented as the schedule permitted.

Some risks from a typical team risk list included

- *Tight schedule.* Risk aversion options included studying the requirements carefully so as not to overcommit, descoping good-to-have features if possible, and concentrating on core capabilities. Risk monitoring activities included closely monitoring all activities to ensure that schedules are met.
- *Project size.* Risk aversion options included descoping good-to-have features and capabilities if requirements were too excessive and identifying the core capabilities to be built.
- *Finding a search engine.* Risk aversion options included conducting a software evaluation of search engines, actively sourcing free search engines for evaluation and selection, and determining the best one for the project. Risk monitoring activities included submitting evaluation reports and conducting demonstrations so that an informed decision can be made.
- *Required technical expertise lacking.* Risk aversion options included identifying the critical and most difficult technical areas of the project and having team members look into them as soon as possible. Monitoring activities included closely following the progress of critical problems and seeking help if necessary.

Client involvement and reaction. The librarians' involvement with the student teams during the second semester was, for the most part, qualitatively and quantitatively different than during the first semester. Major system requirements had already been negotiated, but there were a few new requirements that added subtle differences to the original concepts. Nonetheless, the time required for the librarians' participation was not as extensive as it had been.

Except for one project, the librarians were delighted with the final presentations. Five library clients wanted either to adopt the application as is or extend it for possible adoption. The sixth applica-

tion—the only one not well received—was an attempt to integrate the three photographic-image projects (stereoscopic slides, Hancock Library photo archive, Los Angeles regional history photos) into a single application. The team had only a short time to patch together pieces of three architectures and user interfaces. Some resulting features were good (a colored-glasses stereo capability with good resolution, for example), but none of the clients were enthusiastic about implementing the results.

The librarians expressed that working with Theory W and the WinWin philosophy made it easy for them to "think big" about their projects. The negotiation process balanced that vision by allowing teams and librarians to agree on a feasible set of deliverables for the final products during the academic session. And, although the time commitment was not great, participation in this project let the librarians focus part of their time on multimedia applications and software engineering. One of the greatest advantages for librarians was that they became more familiar with digital library issues and the software engineering techniques involved in their implementation.

Nature of products. As one librarian noted in her evaluation memo

The interaction between the student teams and the librarians produced obvious differences in products designed for different users. For example, the technical reports interface mirrored the technical nature of the type of material included and expected future users of the system, while the student film archive interface reflected the needs and interests of a very different clientele.

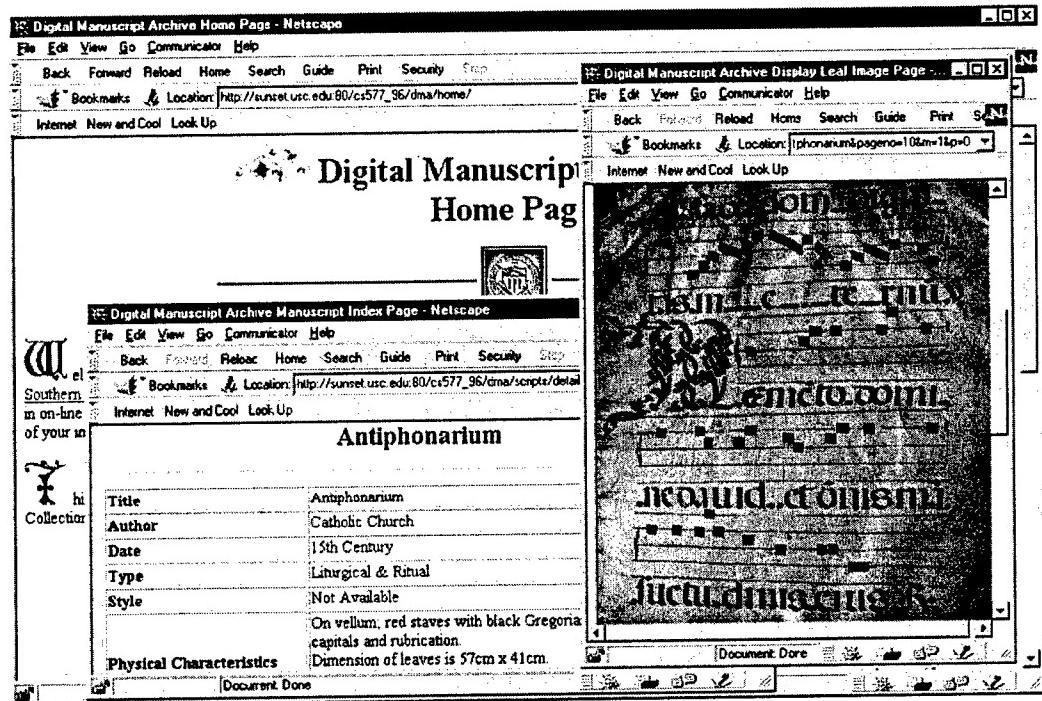
Figure 6, the user interface for the medieval manuscripts application, typifies the look and feel of the products. The Netscape-based application uses various windows to display the manuscript's attributes, to query for desired manuscripts using a search engine, and to enter and catalog new manuscripts.

Adoption of applications. The students spent summer 1997 refining two of the five applications. However, only one of these—the student film archive—was actually implemented. As it turned out, this application was the only one with sufficient budget, people, and facilities to sustain the product after it was implemented. In the following year, we agreed to let the USC library choose which applications would be developed to IOC in spring 1998, and we agreed that we would implement only the applications the client could sustain.

LESSONS LEARNED

When we started the course, we were not sure about any of our choices on such issues as team size, docu-

Figure 6. The user interface for the medieval manuscripts application in Figure 3a. The application satisfies the client's need to scan medieval manuscripts in a way that permits researchers to simultaneously study special markings and read historical data about the image.



ment guidelines, tools, milestones, and course material. However, the library clients and management found the projects sufficiently valuable that they committed both to a continued series of similar projects and to supporting the product's transition and sustaining it after implementation. We, in turn, obtained extensive data and feedback on how to improve the course and project approach both for future courses and for industrial practice.

Number of cycles. For projects of this size, using a single cycle each for the LCO and LCA milestones was about right. Smaller projects can get to the LCA milestone in a single cycle; larger projects may take several cycles to achieve their LCO and LCA goals. Given the results of our LCO reviews, using a single cycle would have produced less satisfactory results in about half the projects. In several projects, the detail was not balanced in either the archiving or query/browsing parts of the LCO packages; the LCA cycle let them correct that imbalance. Using three cycles to produce the LCO and LCA milestones would have left insufficient time to both produce and coordinate three sets of artifacts.

Degree of flexibility. The teams were able to adapt to real-world conditions, such as pleasant and unpleasant surprises with COTS packages, the unavailability of expected infrastructure packages such as the server, lack of expertise on library information systems; and personnel complications. More formal or contract-oriented approaches would not have been able to accommodate these changes in the short time (11 weeks) available.

Communication and trust. We found that, at least for this type of application, the most important outcome of product definition is not a rigorous specification, but a team of stakeholders with enough trust and shared vision to adapt effectively to unexpected

changes. In the beginning, the library clients were considerably uncertain about going forward with the projects. By the LCA milestone, however, the uncertainty and doubt about working with the student teams had been replaced with enthusiasm and considerable trust, although many were still uncertain about the applications' technical parameters. This growth continued through the development period and led to a mutual commitment to pursue additional projects in the following year. The ability of the WinWin approach to foster trust was consistent with earlier experiences.⁶

Smooth transitions. In previous uses of the WinWin spiral model, the transition from WinWin stakeholder agreements to requirements specifications had been rough. The WinWin groupware tool helped smooth this transition. Mapping the WinWin domain taxonomy onto the table of contents of the requirements specification and requiring the use of the domain taxonomy as a checklist for developing WinWin agreements effectively focused stakeholder negotiations. We are exploring how to automate parts of the requirements transition to make it even smoother.

Use of developer time. Although our approach avoided some inefficiencies, we still experienced significant bottlenecks from documentation overkill and attempts to coordinate multiple views. The second-year projects (described later) had less redundant and voluminous documentation, used the Rational Rose integrated object-oriented tool set (which decreased the amount of documentation), and thus yielded fewer inconsistencies. We also had five instead of six members per team, which reduced inconsistencies and overhead because fewer people had to talk to one another.

Finally, we added training and opportunities for feedback. The WinWin groupware tool helped with team building and feature prioritization, but people needed

more preliminary training and experience in its use. Students also cited the need for more training on key Web skills and more feedback on intermediate products. For the second-year projects, we added homework examples both on WinWin principles and preliminary use. We set up special sessions for training on WinWin and Web prototyping. We also set up special LCO and LCA architectural review board sessions for all projects, rather than just three in-class sessions.

Client acceptance. We learned two lessons here. The first is don't finish negotiations before prototyping. If you do, the agreements destabilize once the clients see the prototypes. In the second-year projects, we had the teams negotiate and prototype concurrently. The second lesson is make sure the clients are empowered to support the product not just with knowledge and enthusiasm, but also with resources for the product's operation and maintenance. In the second-year projects, this became our top criterion for selecting applications.

RESULTS OF SECOND-YEAR PROJECTS

In the second-year projects, which we just completed, 16 teams developed similar library-related projects. The process of the second year followed the process of the first year for the most part (from the LCO to LCA to IOC milestones). The changes we made reflected customer and student wishes, their suggestions, and other lessons learned during the first year.

From the 16 projects in the first semester, the clients selected five applications for development according to the library's commitment to sustain them after the second semester (IOC). Four are now transitioning to library operations, and the fifth has good prospects for transition after refinement this summer. We adapted several parts of the first-year process in the second-year projects.

Documentation. We restructured the document guidelines to reduce duplication, and to adapt them for use with Rational Rose and the Unified Modeling Language (we had used OO development and design methods the first year but we did not provide particular tool support for it). The average length of the LCO package decreased from 160 pages in the first-year projects to 103 in the second year.

Layered architectural description. Using UML and Integrated Systems Development Methodology (ISDM)⁷ as our object-oriented methods, we were able to more strongly refine our system software architectural description into three model layers: domain description, system analysis, and system design. Each layer was analogous to the others, but had a different intended audience. The layering improved internal consistency because it maintained distinct relations (documented via simple reference tracing) between the views within and outside each layer. Many teams still found the concepts of consistency and tracing difficult to grasp, but they were more aware than in the first-

year projects that these issues were important. Through the domain description, the teams were able to rapidly understand the parts of their client's domain that were relevant to the target system. With this intermediate representation, the teams were able to work with the client to communicate vital responsibilities, qualities, and components of the target system without losing the client in too much technical design detail. During system analysis, one client commented "I can really see that this [the system] has all the things I expected and is what I wanted." In all, layering helped manage the architectural complexity by letting teams capture, validate, and refine information in a practical and useful way as well as communicate them effectively.

Client acceptance. The extra WinWin and prototyping preparation and training, early prototyping, and adoption of LCO and LCA review boards fit naturally into the WinWin spiral approach and increased the productiveness and quality of the second-year projects over the first-year efforts. There were fewer requirements breakdowns in the later stages of the life cycle, which increased client participation and acceptance to the point of "client activism." Indeed, when a team was given some criticism by the review board, often the client would actively defend the team and their efforts. The resulting discussions often led to identifying additional important objectives, constraints, and alternatives, making the spiral model iterations more effective, and producing more satisfactory products for the clients. The overall satisfaction rating from client critiques, on a scale of 1 to 5, went from 4.3 in the first year to 4.7 in the second.

We are currently addressing improvements for the third year of projects. Of the 80 students in the second-year projects, 26 indicated the need for more UML and Rose education (18 indicated that UML and Rose were very helpful), 13 indicated the need for better document guidelines, and nine indicated the need for a Cocomo II model calibrated to the student projects.

We believe our results so far indicate that the WinWin spiral model will transition well to industry use. The digital library projects were in a sense an industry test because about 20 percent of the teams were purely industry employees, and additional teams had mixes of industry employees and full-time students. In fact, since the first-year projects, industrial organizations have adopted many elements of the WinWin spiral model. Rational, for example, has adopted the LCO, LCA, and IOC definitions as the major milestones in their Objectory or Rational Unified Management Process.^{8,9} MCC is developing an industrial-grade version of the WinWin tool as part of its Software and System

We found that the most important outcome of product definition is not a rigorous specification, but a team of stakeholders with enough trust and shared vision to adapt effectively to unexpected changes.

We believe our results so far indicate that the WinWin spiral model will transition well to industry use.

Acknowledgments

This research is sponsored by DARPA through Rome Laboratory under contract F30602-94-C-0195 and by the affiliates of the USC Center for Software Engineering: Allied Signal, Bellcore, Boeing, Electronic Data Systems, Federal Aviation Administration, GDE Systems, Hughes Aircraft, Interactive Development Environments, Institute for Defense Analysis, Jet Propulsion Laboratory, Litton Data Systems, Lockheed Martin, Loral Federal Systems, MCC, Motorola, Network Programs, Northrop Grumman, Rational Software, Raytheon Science Applications International, Software Engineering Institute, Software Productivity Consortium, Sun Microsystems, TI, TRW, USAF Rome Laboratory, US Army Research Laboratory, and Xerox. We also thank Denise Bedford, Anne Curran, Simei Du, Ellis Horowitz, Ming June Lee, Phil Reese, Bill Scheding, and Nirat Shah for support in key areas.

References

1. B. Boehm and R. Ross, "Theory W Software Project Management: Principles and Examples," *IEEE Trans. Software Eng.*, July 1989, pp. 902-916.
2. B. Boehm et al., "Cost Models for Future Software Processes: COCOMO 2.0," *Annals Software Eng.*, Vol. 1, 1995, pp. 57-94.
3. B. Boehm, "Software Risk Management: Principles and Practices," *IEEE Software*, Jan. 1991, pp. 32-41.
4. I. Sommerville, *Software Engineering*, 5th ed., Addison-Wesley, Reading, Mass., 1996.
5. G. Booch, I. Jacobson, and J. Rumbaugh, "The Unified Modeling Language for Object-Oriented Development," Ver. 1.0, Rational Software Corp., Santa Clara, Calif., 1997.
6. B. Boehm and P. Bose, "A Collaborative Spiral Software Process Model Based on Theory W," *Proc. Int'l Conf. Software Process*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 59-68.
7. D. Port, "Integrated Systems Development Methodology," Telos Press, 1998 (to appear).
8. "Rational Objectory Process," Ver. 4.1, Rational Software Corp., Santa Clara, Calif., 1997.
9. W.E.Royce, *Unified Software Management*, Addison-Wesley, Reading, Mass., 1998 (to be published).

Barry Boehm is the TRW professor of software engineering and director of the Center for Software Engineering at the University of Southern California. His current research involves the WinWin groupware system for software requirements negotiation, architecture-based models of software quality attributes, and the Cocomo II cost-estimation model. Boehm received a PhD in mathematics from the University of California at Los Angeles. He is an AIAA fellow, an ACM fellow, an IEEE fellow, and a member of the National Academy of Engineering.

Alexander Egyed is a PhD student at USC's Center for Software Engineering. His research interests are in software architecture and requirements negotiation. He received an MS in computer science from USC and is a student member of the IEEE.

Julie Kwan is executive and research programs librarian for the Marshall School of Business, Information Services Division at USC. Her interests include information needs and information-seeking behaviors; business, scientific, and technical information transfer; and customer-analysis methodologies. She received an MS in library science from the University of Illinois and is a member of the American Library Association, the Medical Library Association, and the Special Libraries Association.

Dan Port is a research assistant professor at USC and a research associate with the Center for Software Engineering. His primary research interests are in component and object-oriented architectures, systems integration, and partially ordered event structures. He received a PhD in applied mathematics at the Massachusetts Institute of Technology.

Ray Madachy is the manager of the Software Engineering Process Group at Litton Guidance and Control Systems and an adjunct assistant professor of computer science at USC. He received a PhD in industrial and systems engineering from USC. He is a member of the IEEE, ACM, and the International Council in Systems Engineering.

Contact the authors through Egyed at the Center for Software Engineering, USC, Los Angeles, CA 90089-0781; aegyed@sunset.usc.edu.

List of Published Papers and Technical Reports

USC-CSE-95-506 (technical report) published in IEEE Software, March 1996

Aids for Identifying Conflicts Among Quality Requirements

Barry Boehm and Hoh In

One of the biggest risks in software requirements engineering is the risk of over emphasizing one quality attribute requirement (e.g., performance) at the expense of others at least as important (e.g., evolvability and portability). This paper describes an exploratory knowledge-based tool for identifying potential quality attribute risks and conflicts early in the software/system life cycle.

The Quality Attribute Risk and Conflict Consultant (QARCC) examines the quality attribute tradeoffs involved in software architecture and process strategies (e.g., one can improve portability via a layered architecture, but usually at some cost in performance). It operates in the context of the USC-CSE WinWin system, a groupware support system for determining software and system requirements as negotiated win conditions.

USC-CSE-95-507 (technical report) published in IEEE Software, July 1996

Anchoring the Software Process

Barry Boehm

The current proliferation of software process models provides flexibility for organizations to deal with the unavoidably wide variety of software project situations, cultures, and environments. But it weakens their defenses against some common sources of project failure, and leaves them with no common anchor points around which to plan and control. This article identifies three milestones – Life Cycle Objectives, Life Cycle Architecture, and Initial Operational Capability – which can serve as these common anchor points. It also discusses why the presence or absence of these three milestones or their equivalents is a critical success factor, particularly for large software projects, but for other software projects as well.

USC-CSE-96-500 (technical report) published in COMPSAC'96

Software Cost Option Strategy Tool (S-COST)

Barry Boehm and Hoh In

The resolution process of cost conflicts among requirements is complex because of highly collaborative and coordinated processes, complex dependencies, and exponentially increasing option space. This paper describes an exploratory knowledge-based tool (called "S-COST") for assisting stakeholders to surface huge option space and diagnose risks of each option.

The S-COST operates in the context of the USC-CSE WinWin system (a groupware support system for determining software and system requirements as negotiated win conditions), QARCC (a support system for identifying conflicts in quality requirements), and COCOMO (COConstructive COst estimation MOdel).

USC-CSE-96-501 (technical report)

Foundations of the WinWin Requirements Negotiation System

MingJune Lee

Requirements Engineering (RE) constitutes an important part of Software Engineering. The USC WinWin requirements negotiation system addresses critical issues in requirements engineering including (1) multi-stakeholder considerations, (2) change management, and (3) groupware support. The WinWin approach to date has primarily involved exploratory prototyping. The system is now converging on a relatively stable set of artifacts and relationships. This makes it feasible and important to formalize these artifacts and relationships to provide a solid scientific framework for the WinWin system. This is the focused problem addressed by the research presented in this paper.

USC-CSE-96-502 (technical report)

The WinWin Requirements Negotiation System: A Model-Driven Approach

Mingjune Lee and Barry Boehm

Requirements Engineering constitutes an important part of Software Engineering. The USC WinWin requirements negotiation system addresses critical issues in requirements engineering including (1) multi-stakeholder considerations, (2) change management, and (3) groupware support. This paper presents our current research efforts on constructing and reconciling several formal and semi-formal models of the system and its operations, including inter-artifact relationships, artifact life cycles, and equilibrium model. It concentrates on determining the relationships among the various models or views of the WinWin requirements engineering process.

USC-CSE-96-504 (technical report) published in INCOSE'97

Analysis of Software Requirements Negotiation Behavior Patterns

Alexander Egyed and Barry Boehm

Roughly 35 three-person teams played the roles of user, customer, and developer in negotiating the requirements of a library information system. Each team was provided with a suggested set of stakeholder goals and implementation options, but were encouraged to exercise creativity in expanding the stakeholder goals and in creating options for negotiating an eventually satisfactory set of requirements.

The teams consisted of students in a first-year graduate course in software engineering at USC. They were provided with training in the Theory W (win-win) approach to requirements determination and the associated USC WinWin groupware support system. They were required to complete the assignment in two weeks.

Data was collected on the negotiation process and results, with 23 projects providing sufficiently complete and comparable data for analysis. A number of hypotheses were formulated about the results, e.g. that the uniform set of initial conditions would lead to uniform results. This

paper summarizes the data analysis, which shows that expectations of uniform group behavior were generally not realized.

USC-CSE-96-505 (technical report)

Models for Composing Heterogeneous Software Architectures

Ahmed Abd-Allah and Barry Boehm

A persistent problem in software engineering is the problem of software composition. The emergence of software architectures and architectural styles has focused attention on a new set of abstractions with which we can create and compose software systems. We examine the problem of providing a model for the composition of different architectural styles within software systems, i.e. the problem of composing heterogeneous architectures. We describe a model of pure styles that is based on a uniform representation. We provide a disciplined approach for analyzing some key aspects of architectural composition, and show the conditions under which systems will fail to be composed.

USC-CSE-97-501 (technical report)

Extending Reliability Block Diagrams to Software Architectures

Ahmed Abd-Allah

Reliability block diagrams focus on components and connectors as do software architectures. However, some architectural styles possess characteristics which make traditional reliability block diagrams unusable as an analysis technique. In order to use the diagrams, they must be extended to reflect common architectural choices such as concurrency, distribution, dynamism, and implicit connectors.

USC-CSE-97-502 (technical report)

Detecting Architectural Mismatches During Systems Composition—An Extension to the AAA Model

Cristina Gacek

The USC Architect's Automated Assistant (AAA) tool and method provides a capability for early detection of architectural style mismatches among four architectural styles: Main-Subroutine, Pipe-and-Filter, Event-Based, and Distributed Processes.

The work proposed here is to formalize some additional architectural styles—namely Blackboard, Closed-Loop Feedback Control, Logic Programming, Real-Time, Rule-Based, and Transactional Database styles—and to extend the mismatch analysis capability to cover interactions of the original four styles with the new ones. The application of the mismatch analysis capability to a relevant problem will also be included in the future.

IEEE Software, May/June 1997, pp. 17-19

Software Risk Management

Barry Boehm and Tom DeMarco

In mature disciplines, risk management has been *de rigueur* for centuries. When Michelangelo set out to raise the dome of St. Peters in 1547, he was well aware of the potential collapse zones under staging, the possibility of materials failure, and the human capacity for error. For each of these major risks he prepared a mitigation plan: a fallback, a safety factor, or an alternative.

Today, we routinely practice risk management in our stewardship of the environment, in planning financial strategy, in construction engineering, and in medicine. But how do we apply it to the ultimate risky business, software development?

USC-CSE-97-503 (technical report)

WinWin Reference Manual—A System for Collaboration and Negotiation

Ellis Horowitz

WinWin is a computer program that aids in the capture, negotiation, and coordination of requirements for a large system. It assumes that a group of people, called stakeholders, have signed on with the express purpose of discussing and refining the requirements of their proposed system. The system can be of any type. This is the 1999 update of the original WinWin Reference Manual.

USC-CSE-97-504 (technical report) published at ESEC/ FSE'97

Developing Multimedia Applications with the WinWin Spiral Model

Barry Boehm, Alex Egyed, USC-Center for Software Engineering

Julie Kwan, USC University Libraries

Ray Madachy, USC-CSE and Litton Data Systems

Fifteen teams recently used the WinWin Spiral Model to perform the system engineering and architecting of a set of multimedia applications for the USC Library Information Systems. Six of the applications were then developed into an Initial Operational Capability. The teams consisted of USC graduate students in computer science. The applications involved extensions of USC's UNIX-based, text-oriented, client-server Library Information System to provide access to various multimedia archives (films, videos, photos, maps, manuscripts, etc.).

Each of the teams produced results which were on schedule and (with one exception) satisfactory to their various Library clients. This paper summarizes the WinWin Spiral Model approach taken by the teams, the experiences of the teams in dealing with project challenges, and the major lessons learned in applying the Model. Overall, the WinWin Spiral Model provided sufficient flexibility and discipline to produce successful results, but several improvements were identified to increase its cost-effectiveness and range of applicability.

USC-CSE-97-506 (technical report)

Detecting Architectural Mismatches During Systems Composition

Cristina Gacek

The USC Architect's Automated Assistant (AAA) tool and method provides a capability for early detection of architectural style mismatches among four architectural styles: Main-Subroutine, Pipe-and-Filter, Event-Based, and Distributed Processes. For these four styles, mismatch detection is based on a set of seven conceptual features distinguishing each style, and a set of eight types of bridging connectors characterizing compositions among the four styles.

The work proposed here is to formalize some additional architectural styles—namely Blackboard, Closed-Loop Feedback Control, Logic Programming, Real-Time, Rule-Based, and Transactional Database styles—and to extend the mismatch analysis capability to cover interactions of the original four styles with the new ones. The analysis results will test various hypotheses, such as the sufficiency of the original seven conceptual features and eight bridging connector types to characterize the broader set of styles and their composition.

We will also try to provide a more formal basis for detecting and classifying architectural conceptual features, thus providing a formal framework for extending the models. The application of the broadened mismatch analysis capability to a relevant problem will also be included in the future.

USC-CSE-97-508 (technical report) published at ICSP'98

WinWin Requirements Negotiation Processes: A Multi-Project Analysis

Barry Boehm and Alexander Egyed

Fifteen 6-member-teams were involved in negotiating requirements for multimedia software systems for the Library of the University of Southern California. The requirements negotiation used the Stakeholder WinWin success model and the USC WinWin negotiation model (Win Condition-Issue-Option-Agreement) and groupware system. The negotiated results were integrated into a Life Cycle Objectives (LCO) package for the project, including descriptions of the system's requirements, operational concept, architecture, life cycle plan, and feasibility rationale. These were subsequently elaborated into a Life Cycle Architecture package including a prototype; six of these were then implemented as products.

The software engineers involved were computer science students (mostly first year graduate level) at USC and the customers were librarians of the USC Library. Their set of problems were very diverse, having to do with Medieval Manuscripts, Student Films, Corporate Business Data, Stereoscopic Slides, and more, each project with its unique constraints and desired capabilities.

A number of hypotheses were tested regarding the effectiveness of the WinWin approach in supporting the development of effective LCO packages, in satisfying Library clients, and in stimulating cooperation among stakeholders. Other hypotheses involved identification of WinWin improvements, relationships among negotiation strategies on LCO package and project outcomes.

Some of the more illuminating results were:

- Most of the stakeholder Win Conditions were non-controversial (were not involved in Issues). Also, most Issues were decoupled from other Issues and were easy to resolve. This implies that requirements negotiation support systems should focus at least as much on handling simple relationships well as on handling complex relationships well.
 - Negotiation activity varied by stakeholder role. Users and customers were more active in the early stages; developers and customers in the late stages.
 - LCO package strength and consistency (measured by LCO grading criteria) could be predicted from three attributes (team experience, non-sequential negotiation, and efficiency in producing negotiation artifacts).
 - The strongest positive effects of using the WinWin approach were increasing cooperativeness, focusing participants on key issues, reducing friction, and facilitating distributed collaboration.
 - The major improvements for the WinWin approach (now being implemented) were increasing WinWin training, reducing usage overhead, and concurrent negotiation and prototyping.
-

**USC-CSE-97-509 (technical report) published at ISPW 10
Process Support of Software Product Lines (ISPW 10)**

Barry Boehm

The focus of ISPW 10 was on “Process Support of Software Product Lines.” Much of the technology currently available to support the software process has focused on the process of developing and evolving a single software product. Increasingly, organizations are finding advantages in product-line software approaches, involving investments in domain engineering, product line architectures, and rapid applications composition with extensive use of commercial-off-the-shelf (COTS) and other reusable software assets. Recent books on software reuse and product line management provide extensive evidence of the advantages: factors of 1.5 to 4 improvements in development time, factors of 1.5 to 6 in productivity, and factors of 2-10 in defect rates. This paper summarizes the original issues and major conclusions of the Workshop.

**USC-CSE-98-500 (technical report) published at INCOSE'98
A Comparison Study in Software Requirements Negotiation**

Alexander Egyed and Barry Boehm

In a period of two years, two rather independent experiments were conducted at the University of Southern California. In 1995, 23 three-person teams negotiated the requirements for a hypothetical library system. Then in 1996, 14 six-person teams negotiated the requirements for real multimedia related library systems.

A number of hypotheses were created to test how real software projects differ from hypothetical ones. Other hypotheses address differences in uniformity and repeatability.

The results indicate that repeatability in 1996 was even harder to achieve than in 1995 (Egyed-Boehm, 1996). Nevertheless, this paper presents some surprising commonalities between both years that indicate some areas of uniformity.

In both years, the same overall development process (spiral model) was followed, the same negotiation tools (WinWin System) were used, and the same people were doing the analysis of the findings. Thus, the comparison is less blurred by fundamental differences like terminology, process, etc.

USC-CSE-98-501 (technical report) published at ICSE'98
Software Requirements Negotiation: Some Lessons Learned

Barry Boehm and Alexander Egyed

Negotiating requirements is one of the first steps in any software system life cycle, but its results have probably the most significant impact on the system's value. However, the processes of requirements negotiation are not well understood. We have had the opportunity to capture and analyze requirements negotiation behavior for groups of projects developing library multimedia archive systems, using an instrumented version of the USC WinWin groupware system for requirements negotiation. Some of the more illuminating results were:

- Most stakeholder Win Conditions were non-controversial (were not involved in Issues)
 - Negotiation activity varied by stakeholder role.
 - LCO package quality (measured by grading criteria) could be predicted by negotiation attributes.
 - WinWin increased cooperativeness, reduced friction, and helped focus on key issues.
-

USC-CSE-98-505 (technical report) published the OMG-DARPA-MCC Workshop on Compositional Software Architectures

Composing Components: How Does One Detect Potential Architectural Mismatches?

Cristina Gacek and Barry Boehm

Nowadays, in order to be competitive, a developer's usage of Commercial off the Shelf (COTS), or Government off the Shelf (GOTS), packages has become a sine qua non, at times being an explicit requirement from the customer. The idea of simply plugging together various COTS packages and/or other existing parts results from the megaprogramming principles. What people tend to trivialize is the side effects resulting from the plugging or composition of these subsystems. Some COTS vendors tend to preach that because their tool follows a specific

standard, say CORBA, all composition problems disappear. Well, it actually is not that simple. Side effects resulting from the composition of subsystems are not just the result of different assumptions in communication methods by various subsystems, but the result from differences in various sorts of assumptions, such as the number of threads that are to execute concurrently, or even on the load imposed on certain resources. This problem is referred to as architectural mismatches. Some but not all of these architectural mismatches can be detected via domain architecture characteristics, such as mismatches in additional domain interface types (units, coordinate systems, frequencies), going beyond the general interface types in standards such as CORBA.

Other researchers have successfully approached reuse at the architectural level by limiting their assets not by domain, but rather by dealing with a specific architectural style. I.e., they support reuse based on limitations on the architectural characteristics of the various parts and resulting systems. This approach can be successful because it simply avoids the occurrence of architectural mismatches.

Our work addresses the importance of underlying architectural features in determining potential architectural mismatches while composing arbitrary components. We have devised a set of those features, which we call conceptual features, and are building a model that uses them for detecting potential architectural mismatches. This underlying model has been built using Z.

USC-CSE-98-507 (technical report) published IFIP'98
Telecooperation Experience with the WinWin System
Alexander Egyed and Barry Boehm

WinWin is a telecooperation system supporting the definition of software-based applications as negotiated stakeholder win conditions. Our experience in using WinWin in defining over 30 digital library applications, including several telecooperation systems, is that it is important to supplement negotiation support systems such as WinWin with such capabilities as prototyping, tradeoff analysis tools, email, and videoconferencing. We also found that WinWin's social orientation around considering other stakeholders' win conditions has enabled stakeholders to achieve high levels of shared vision and mutual trust. Our subsequent experience in implementing the specified digital library systems in a rapidly changing web-based milieu indicated that achieving these social conditions among system stakeholders was more important than achieving precise requirements specifications, due to the need for team adaptability to requirements change. Finally, we found that the WinWin approach provides an effective set of methods of integrating ethical considerations into practical system definition processes via Rawls' stakeholder negotiation-based Theory of Justice.

USC-CSE-98-509 (technical report) published at EUROMICRO'98
Improving the Life-Cycle Process in Software Engineering Education
Barry Boehm and Alexander Egyed

The success of software projects and the resulting software products are highly dependent on the initial stages of the life-cycle process – the inception and elaboration stages. The most critical success factors in improving the outcome of software projects have often been identified as being the requirements negotiation and the initial architecting and planning of the software system. Not surprisingly, this area has thus received strong attention in the research community. It has, however, been hard to validate the effectiveness and feasibility of new or improved concepts because they are often only shown to work in a simplified and hypothesized project environment. Industry, on the other hand, has been cautious in adopting unproven ideas. This has led to a form of deadlock between those parties.

In the last two years, we had had the opportunity to observe dozens of software development teams in planning, specifying and building library related, real-world applications. This environment provided us with a unique way of introducing, validating and improving the life cycle process with new principles such as the WinWin approach to software development. This paper summarizes the lessons we have learned.

USC-CSE-98-510 (technical report) published in WICSA'99

The MBASE Life Cycle Architecture Milestone Package: No Architecture Is An Island

Barry Boehm, Dan Port, Alexander Egyed, Marwan Abi-Antoun

This paper summarizes the primary criteria for evaluating software/system architectures in terms of key system stakeholders' concerns. It describes the Model Based Architecting and Software Engineering (MBASE) approach for concurrent definition of a system's architecture, requirements, operational concept, prototypes, and life cycle plans. It summarizes our experiences in using and refining the MBASE approach on 31 digital library projects. It concludes that a Feasibility Rationale demonstrating consistency and feasibility of the various specifications and plans is an essential part of the architecture's definition, and presents the current MBASE annotated outline and guidelines for developing such a Feasibility Rationale.

USC-CSE-98-511 (technical report) published in Annals of Software Engineering, 1999

A Stakeholder Win-Win Approach to Software Engineering Education

Barry Boehm, Alexander Egyed, Dan Port, and Archita Shah , USC-Center for Software Engineering, Julie Kwan, USC University Libraries and Ray Madachy, USC-CSE and Litton Data Systems

We are applying the stakeholder win-win approach to software engineering education. The key stakeholders we are trying to simultaneously satisfy are the students; the industry recipients of our graduates; the software engineering community as parties interested in improved practices; and ourselves as instructors and teaching assistants. In order to satisfy the objectives or win conditions of these stakeholders, we formed a strategic alliance with the University of Southern California Libraries to have software engineering student teams work with Library clients to define, develop, and transition USC digital library applications into operational use. This adds another set of key stakeholders: the Library clients of our class projects. This paper summarizes our experience in developing, conducting, and iterating the course. It concludes by evaluating the degree to which we have been able to meet the stakeholder-determined course objectives.

USC-CSE-98-512 (technical report) published in IEEE Computer, July 1998

Using the WinWin Spiral Model: A Case Study

Barry Boehm, Alexander Egyed, Julie Kwan, Dan Port, and Archita Shah, USC-Center for Software Engineering and Ray Madachy, USC-CSE and Litton Data Systems

Fifteen teams used the WinWin spiral model to prototype, plan, specify, and build multimedia applications for USC's Integrated Library System. The authors report lessons learned from this case study and how they extended the model's utility and cost-effectiveness in a second round of projects.

USC-CSE-98-513 (technical report) published in Proceedings, Conceptual Modeling Symposium
Conceptual Modeling Challenges for Model-Based Architecting and Software Engineering (MBASE)

Barry Boehm and Dan Port

The difference between failure and success in developing a software-intensive system can often be traced to the presence or absence of clashes among the models used to define the system's product, process, property, and success characteristics. (Here, we use a simplified version of one of Webster's definitions of "model" a description or analogy used to help visualize something. We include analysis as a form of visualization).

Section 2 of this paper introduces the concept of model clashes, and provides examples of common clashes for each combination of product, process, property, and success models. Section 3 introduces the Model-Based Architecting and Software Engineering (MBASE) approach for endowing a software project with a mutually supportive base of models. Section 4 presents examples of applying the MBASE approach to a family of digital library projects.

Section 5 summarizes the main conceptual modeling challenges involved in the MBASE approach, including integration of multiple product views and integration of various classes of product, process, property, and success models. Section 6 summarizes current conclusions and future prospects.

USC-CSE-98-514 (technical report) published HICSS'99

Rose/Architect: a tool to visualize architecture

Alexander Egyed and Philippe Kruchten

Rational Rose is a graphical software modeling tool, using the Unified Modeling Language (UML) as its primary notation. It offers an open API that allows the development of additional functionality ("add-ins"). In this paper, we describe Rose/Architect, a Rose™ "add-in" used to visualize architecturally-significant elements in a system's design, developed jointly by University of Southern California (USC) and Rational Software. Rose/Architect can be used in forward engineering, marking architecturally significant elements as they are designed and

extracting architectural views as necessary. But it can be even more valuable in reverse engineering, i.e., extracting missing key architectural information from a complex model. This model may have been reverse-engineered from source code using the Rose reverse engineering capability.

USC-CSE-98-517 (technical report) published in Software Engineering Notes, 1999

Escaping the Software Tar Pit: Model Clashes and How to Avoid Them

Barry Boehm and Dan Port

"No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits... Large system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it..."
"Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it."
Fred Brooks, 1975

Several recent books and reports have confirmed that the software tar pit is at least as hazardous today as it was in 1975. Our research into several classes of models used to guide software development (product models, process models, property models, success models), has convinced us that the concept of model clashes among these classes of models helps explain much of the stickiness of the software tar-pit problem.

We have been developing and experimentally evolving an approach called MBASE -- Model-Based (System) Architecting and Software Engineering -- which helps identify and avoid software model clashes. Section 2 of this paper introduces the concept of model clashes, and provides examples of common clashes for each combination of product, process, property, and success model. Sections 3 and 4 introduce the MBASE approach for endowing a software project with a mutually supportive set of models, and illustrate the application of MBASE to an example corporate resource scheduling system. Section 5 summarizes the results of applying the MBASE approach to a family of small digital library projects. Section 6 presents conclusions to date.

USC-CSE-98-518 (technical report) published in ICRE'99

Requirements Engineering, Expectations Management, and The Two Cultures

Barry Boehm, Marwan Abi-Antoun, and Dan Port, USC-CSE,

Julie Kwan, USC University Libraries,

Anne Lynch, University of Southern California

In his seminal work, *The Two Cultures*, C.P. Snow found that science and technology policymaking was extremely difficult because it required the combined expertise of both scientists and politicians, whose two cultures had little understanding of each other's principles and practices.

During the last three years, we have conducted over 50 real-client requirements negotiations for digital library applications projects. Those largely involve professional librarians as clients

and 5-6 person teams of computer science MS-degree students as developers. We have found that their two-cultures problem is one of the most difficult challenges to overcome in determining a feasible and mutually satisfactory set of requirements for these applications.

During the last year, we have been experimenting with expectations management and domain-specific lists of "simplifiers and complicators" as a way to address the two-cultures problem for software requirements within the overall digital library domain. Section 2 of this paper provides overall motivation and context for addressing the two-cultures problem and expectations management as significant opportunity areas in requirements engineering. Section 3 discusses the digital library domain and our stakeholder Win-Win and Model-Based (System) Architecting and Software Engineering (MBASE) approach as applied to digital library projects. Section 4 discusses our need for better expectations management in determining the requirements for the digital library projects are products over the first two years, and describes our approach in year 3 to address the two-cultures problem via expectations management. Section 5 summarizes results to date and future prospects.

USC-CSE-98-519 (technical report)

Guidelines for the Life Cycle Objectives (LCO) and the Life Cycle Architecture (LCA) deliverables for Model-Based Architecting and Software Engineering (MBASE)

Barry Boehm, Dan Port, Marwan Abi-Antoun and Alexander Egyed

Over our three years of developing digital library products for the USC Libraries, we have been evolving an approach called Model-Based (System) Architecting and Software Engineering (MBASE). MBASE involves early reconciliation of a project's success models, product models, process models, and property models. It extends the previous spiral model in two ways:

Initiating each spiral cycle with a stakeholder win-win stage to determine a mutually satisfactory (win-win) set of objectives, constraints, and alternatives for the system's next elaboration during the cycle.

Orienting the spiral cycles to synchronize with a set of life cycle anchor points: Life Cycle Objectives (LCO), Life Cycle Architecture (LCA), and Initial Operational Capability (IOC). The MBASE guidelines present the content and the completion criteria for the LCO and LCA milestones (which correspond to the Inception and Elaboration Phases of the Rational Unified Process) of the following system definition elements:

- Operational Concept Description (OCD)
- System and Software Requirements Definition (SSRD)
- System and Software Architecture Description (SSAD)
- Life Cycle Plan (LCP)
- Feasibility Rationale Description (FRD)
- Risk-driven prototypes

The guidelines also include a suggested domain taxonomy to be used as a checklist and organizing structure for the WinWin requirements negotiation. The guidelines attempt to achieve

high conceptual integrity, little redundancy, and strong traceability across the various system definition elements, and are compatible with the Unified Modeling Language (UML).

These guidelines were used by 20 teams of 5-6 person teams of computer science graduate students during Fall 1998, and were revised twice following the LCO and LCA Architecture Review Boards. These guidelines were used for rebaselining the LCA packages during Spring 99.

USC-CSE-99-511 (technical report)

Automating Architectural View Integration in UML

Alexander Egyed

Architecting software systems requires more than what general-purpose software development models can provide. Architecting is about modeling, solving and interpreting, and in doing so, placing a major emphasis on mismatch identification and reconciliation within and among architectural views (such as diagrams). The emergence of the Unified Modeling Language (UML), which has become a de-facto standard for OO software development, is no exception to that. This work describes causes of architectural mismatches for UML views and shows how integration techniques can be applied to identify and resolve them in a more automated fashion.

USC-CSE-99-512 (technical report) published in part in IEEE Computer, 3/99

Making RAD Work for Your Project

Barry Boehm

A significant recent trend we have observed among our USC Center for Software Engineering's industry and government Affiliates is that reducing the schedule of a software development project was becoming considerably more important than reducing its cost. This led to an Affiliates' Workshop on Rapid Application Development (RAD) to explore its trends and issues. Some of the main things we learned at the workshop were:

- There are good business reasons why software development schedule is often more important than cost.
 - There are various forms of RAD. None are best for all situations. Some are to be avoided in all situations.
 - For mainstream software development projects, we could construct a RAD Opportunity Tree which helps sort out the best RAD mixed strategy for a given situation.
-

USC-CSE-99-515 (technical report)

Using Patterns to Integrate UML Views

Alexander Egyed

Patterns play a major role during system composition (synthesis) in fostering the reuse of repeatable design and architecture configurations. This paper investigates how knowledge about patterns may also be used for system analysis to verify the conceptual integrity of the system model.

To support an automated analysis process, this work introduces a view integration framework. Since each view (e.g. diagram) adds an additional perspective of the software system to the model, information from one view may be used to validate the integrity of other views. This form of integration requires a deeper understanding as to what the views mean and what information they can share (or constrain). Knowledge about patterns, both in structure and behavior, are thereby a valuable source for view integration automation.

USC-CSE-99-516 (technical report)

Enabling Distributed Collaborative Prioritization

Daniel Port and Jung-Won Park

One of the most common problems within a risk driven software collaborative development effort is prioritizing items such as requirements, goals, and stakeholder win-conditions. Requirements have proven particularly sticky in this as it is often the case that they can not be fully implemented when time and resources are limited introducing additional risk to the project. A practical approach to mitigating this risk in alignment with the WinWin development approach is to have the critical stakeholders for the project collaboratively negotiate requirements into priority bins which then are scheduled into an appropriate incremental development life cycle.

We have constructed a system called the Distributed Collaboration Priorities Tool (DCPT) which to assist in collaborative prioritization of development items. DCPT offers a strcutually guided approach to collaborative prioritization much in the spirit of USC's WinWin requirements capture and negotiation system. In this paper, we will discuss the prioritization models implemented within DCPT via an actual prioritization of new WinWin system features. We also discuss DCPT's two-way integration with WinWin system, some experiences using DCPT, and current research directions.

USC-CSE-99-520 (technical report) published in Journal for Computer Standards and Interfaces

Optimizing Software Product Integrity through Life-Cycle Process Integration

Barry Boehm and Alexander Egyed

Managed and optimized - these are the names for the levels 4 and 5 of the Capability Maturity Model (CMM) respectively. With that the Software Engineering Institute (SEI) pays tribute to the fact that, after the process has been defined, higher process maturity, and with that higher product maturity, can only be achieved by improving and optimizing the life-cycle process itself. In the last three years, we had had the opportunity to observe more than 50 software development teams in planning, specifying and building library related, real-world applications. This environment provided us with a unique way of introducing, validating and improving the life cycle process with new principles such as the WinWin approach to software development.

This paper summarizes the lessons we have learned in our ongoing endeavor to integrate the WinWin life-cycle process. In doing so, we will not only describe what techniques have proven to be useful in getting the developer's task done but the reader will also get some insight on how to tackle process improvement itself. As more and more companies are reaching CMM levels two or higher this task, of managing and optimizing the process, becomes increasingly important.

USC-CSE-99-521 (technical report) published in Journal for Systems Engineering

Comparing Software System Negotiation Requirements Patterns

Alexander Egyed and Barry Boehm

In a period of two years, two rather independent experiments were conducted at the University of Southern California (USC). In 1995, 23 three-person teams negotiated the requirements for a hypothetical library system. Then, in 1996, 14 six-person teams negotiated the requirements for real-world digital library systems.

A number of hypotheses were created to test how more realistic software projects differ from hypothetical ones. Other hypotheses address differences in uniformity and repeatability of negotiation processes and results. The results indicate that repeatability in 1996 was even harder to achieve than in 1995. Nevertheless, this paper presents some surprising commonalities between both years that indicate some areas of uniformity.

As such we found that the more realistic projects required more time to resolve conflicts and to identify options (alternatives) than the hypothetical ones. Further, the 1996 projects created more artifacts although they exhibited less artifact interconnectivity, implying a more divide and conquer negotiation approach. In terms of commonalities, we found that people factors such as experience did have effects onto negotiation patterns (especially in 1996), that users and customers were most significant (in terms of artifact creation) during the goal identification whereas the developers were more significant in identifying issues (conflicts) and options. We also found that both years exhibited some strange although similar disproportional stakeholder participation.

USC-CSE-99-522 (technical report) published in ICSE'99

WinWin: a System for Negotiating Requirements

Ellis Horowitz, Joo H. Lee, and June Sup Lee

WinWin is a system that aids in the capture and recording of system requirements. It also assists in negotiation. The WinWin system has been available for several years and it being used by dozens of software development groups. In this presentation we will go over the capabilities of the system and discuss how it might be used on your software development project.

USC-CSE-99-523 (technical report) published in IEEE IT Professional, Jan-Feb 1999

When Models Collide: Lessons From Software System Analysis

Barry Boehm and Dan Port

This paper analyzes several classes of model clashes encountered on large, failed IT projects (e.g., Confirm, Master Net), and shows how the MBASE approach could have detected and resolved the clashes.

USC-CSE-99-526 (technical report) published in FASE, May 2000.

A Formal Approach to Heterogeneous Software Modeling

Alexander Egyed and Nenad Medvidovic

The problem of consistently engineering large, complex software systems of today is often addressed by introducing new, “improved” models. Examples of such models are architectural, design, structural, behavioral, and so forth. Each software model is intended to highlight a particular view of a desired system. A combination of multiple models is needed to represent and understand the entire system. Ensuring that the various models used in development are consistent relative to each other thus becomes a critical concern. This paper presents an approach that integrates and ensures the consistency across an architectural and a number of design models. The goal of this work is to combine the respective strengths of a powerful, specialized (architecture-based) modeling approach with a widely used, general (design-based) approach. We have formally addressed the various details of our approach, which has allowed us to construct a large set of supporting tools to automate the related development activities. We use an example application throughout the paper to illustrate the concepts.

USC-CSE-99-527 (technical report) published in IWSAPF 2000

Software Connectors and Refinement in Product Families

Alexander Egyed, Nikunj Mehta, and Nenad Medvidovic

Product families promote reuse of software artifacts such as architectures, designs and implementations. Product family architectures are difficult to create due to the need to support variations. Traditional approaches emphasize the identification and description of generic components which prove too rigid to support variations in each product. This paper presents an approach that supports analyzable family architectures using generic software connectors that provide bounded ambiguity and support flexible product families. It describes the transformation from a family architecture to a product design through a four-way refinement and evolution process

USC-CSE-99-529 (technical report) published in ICSE 2000

Towards a Taxonomy of Software Connectors

Nikunj Mehta, Nenad Medvidovic and Sandeep Phadke

Software systems of today are frequently composed from prefabricated, heterogeneous components that provide complex functionality and engage in complex interactions. Existing research on component-based development has mostly focused on component structure, interfaces, and functionality. Recently, software architecture has emerged as an area that also places significant importance on component interactions, embodied in the notion of software connectors. However, the current level of understanding and support for connectors has been insufficient. This has resulted in their inconsistent treatment and a notable lack of understanding of what the fundamental building blocks of software interaction are and how they can be

composed into more complex interactions. This paper attempts to address this problem. It presents a comprehensive classification framework and taxonomy of software connectors. The taxonomy is used both to understand existing software connectors and to suggest new, unprecedented connectors. We demonstrate the use of the taxonomy on the architecture of an existing, large system.

USC-CSE-2000-500 (technical report)

Why Consider Implementation-Level Decisions in Software Architectures?

Nikunj Mehta, Nenad Medvidovic and Marija Rakic

Software architecture provides a high-level abstraction of the structure, behavior, and properties of a software system aimed at enabling early analysis of the system and its easier implementation. Often, however, important details about a system are left to be addressed in its implementation, resulting in differences between conceptual and concrete architectures. This paper describes an approach towards bringing these two closer by making certain implementation-level decisions explicit in the architecture. Specifically, we focus on the choices made in modeling and implementing component interactions. The process is based on a taxonomy of software connectors that the authors have developed to better understand component interactions, and an architectural framework developed to support a variety of connectors.

USC-CSE-2000-507 (technical report)

Spiral Development: Experience, Principles, and Refinements

Barry Boehm. USC Center for Software Engineering

This presentation opened the USC-SEI Workshop on Spiral Development* Experience and Implementation Challenges held at USC February 9-11, 2000. The workshop brought together leading executives and practitioners with experience in transitioning to spiral development of software-intensive systems in the commercial, aerospace, and government sectors. Its objectives were to distill the participants' experiences into a set of critical success factors for transitioning to and successfully implementing spiral development, and to identify the most important needs, opportunities, and actions to expedite organizations' transition to successful spiral development. To provide a starting point for addressing these objectives, I tried in this talk to distill my experiences in developing and transitioning the spiral model at TRW; in using it in system acquisitions at DARPA; in trying to refine it to address problems that people have had in applying it in numerous commercial, aerospace, and government contexts; and in working with the developers of major elaborations and refinements of the spiral model such that the Software Productivity Consortium's Evolutionary Spiral Process and Rational, Inc's Rational Unified Process. I've modified the presentation somewhat to reflect the experience and discussions at the Workshop.

USC-CSE-2000-508 (technical report) published in IEEE Computer 2000

Managing Software Productivity and Reuse

Barry Boehm

USC-CSE-2000-509 (technical report) published in UML 1999

Extending Architectural Representation in UML with View Integration

Alexander Egyed and Nenad Medvidovic

UML has established itself as the leading OO analysis and design methodology. Recently, it has also been increasingly used as a foundation for representing numerous (diagrammatic) views that are outside the standardized set of UML views. An example are architecture description languages. The main advantages of representing other types of views in UML are 1) a common data model and 2) a common set of tools that can be used to manipulate that model. However, attempts at representing additional views in UML usually fall short of their full integration with existing views. Integration extends representation by also describing interactions among multiple views, thus capturing the inter-view relationships. Those inter-view relationships are essential to enable automated identification of consistency and conformance mismatches. This work describes a view integration framework and demonstrates how an architecture description language, which was previously only represented in UML, can now be fully integrated into UML.

USC-CSE-2000-510 (technical report) published in ASE 2000

Automatically Detecting Mismatches during Component-Based and Model-Based Development

Alexander Egyed and Cristina Gacek

A major emphasis in software development is placed on identifying and reconciling architectural and design mismatches. Those mismatches happen during software development on two levels: while composing system components (e.g. COTS or in-house developed) and while reconciling view perspectives. Composing components into a system and 'composing' views (e.g. diagrams) into a system model are often seen as being somewhat distinct aspects of software development, however, as this work shows, their approaches in detecting mismatches complement each other very well. In both cases, the composition process may result in mismatches that are caused by clashes between development artefacts. Our component-based integration approach is more high-level and can be used early on for risk assessment while little information is available. Model-based integration, on the other hand needs more information to start with but is more precise and can handle large amounts of redundant information. This paper describes both integration approaches and discusses their commonalities and differences. Both integration approaches are automateable and some tools support is already available.

USC-CSE-2000-511 (technical report)

Automated Abstraction for Object-Oriented Models

Alexander Egyed

Working with multiple views allows development concerns to be broken up and investigated separately, thus, reducing software development complexity. On the downside, views also require an explicit notion on how to exchange information among them – a necessity caused by the fact that problems and solutions described in views have to be integrated with one another to form a coherent and consistent whole. This paper discusses a method for automated abstraction of

diagrammatic views with a major emphasis on class and object diagrams. This technique is also suited for consistency checking and reverse engineering. Our approach is fully tool supported and we have since validated both tool and model through a series of experiments.

USC-CSE-2000-512 (technical report)

Using Model Transformations to Detect Inconsistencies between Heterogeneous Views

Alexander Egyed

Development is about modeling, solving and interpreting, and in doing so a major emphasis is placed on mismatch identification and reconciliation within and among diagrammatic and textual views. It has been acknowledged that view integration techniques generally do not scale because of the complexities involved. In response, it has been proposed to use transformation techniques to simplify comparison. However, what has been only little explored is the fact that view transformation introduces new types of scalability problems reducing, if not worsening, their benefits towards view integration. This work introduces a view integration framework and demonstrates how transformation can enable view comparison in a more scalable and reliable fashion. We will discuss view integration in the context of the Unified Modeling Language (UML) where we will show, on concrete examples, how transformation simplifies view integration and how the resulting scalability problems can be addressed.

USC-CSE-2000-513 (technical report)

Software Lifecycle Connectors: Bridging Models across the Lifecycle

Nenad Medvidovic, Paul Gruenbacher, Alexander Egyed, and Barry Boehm

Numerous notations, methodologies, and tools exist to support software system modeling. While individual models may clarify certain system aspects, the large number and heterogeneity of models may ultimately hamper the ability of stakeholders to communicate about a system. The major reason for this is the discontinuity of information across different models. In this paper, we present an approach for dealing with that discontinuity. We propose an extensible set of "connectors" to bridge models, both within and across the activities in the software development lifecycle. While the details of these connectors are dependent upon the source and destination models, they share a number of underlying characteristics. We illustrate our approach by applying it to a large-scale system we are currently designing and implementing in collaboration with a third-party organization.

USC-CSE-2000-514 (technical report)

Architectural Integration and Evolution in a Model World

Alexander Egyed (USC-CSE) and Rich Hilliard (ISIS 2000)

Architectural Description Languages (ADLs) fall into the narrow category of frequently using only one fixed representation scheme. Over the past years, it has become more obvious that no single such ADL is adequate in addressing a large number of stakeholder concerns. This paper, therefore, discusses the need and challenges of multi-view development with ADLs and introduces a decorative stance in handling view integration issues and scalability concerns related to consistency checking.

USC-CSE-2000-515 (technical report)

Refinement and Evolution Issues between Requirements and Architecture

Alexander Egyed, Nenad Medvidovic, and Paul Gruenbacher

Though acknowledged as very closely related to a large extent, requirements engineering and architecture modeling have been pursued independently of one another, particularly in the large body of software architecture research that has emerged over the past decade. The dependencies and constraints imposed by elements of one on those of the other are not well understood. This paper identifies a number of relevant relationships we have identified in the process of trying to relate the WinWin requirements engineering approach with architecture and design-centered approaches (e.g., C2 and UML).

USC-CSE-2000-516 (technical report)

An Integrated Perspective on Software Mismatch Detection and Resolution

Alexander Egyed, Nenad Medvidovic, and Cristina Gacek

Modeling software systems all too often neglects the issue of mismatch identification and resolution. The traditional view of modeling over-emphasizes synthesis at the expense of analysis - the latter frequently being seen as a problem one only needs to deal with during the integration stage towards the end of a development project. This paper discusses three software modeling and analysis techniques, all tool supported, and emphasizes the vital role analysis can play in identifying and resolving risks early on. This work also combines model based development with component based development (e.g., COTS and Legacy systems) and shows how their mismatch detection capabilities complement each other in providing a more comprehensive coverage of development risks.

USC-CSE-2000-517 (technical report)

A Scenario-Driven Approach to Traceability

Alexander Egyed (USC-CSE)

Software engineering uses models and views to handle software development concerns. The major drawback of a model-based development approach is that concerns cannot be investigated individually, since they tend to affect each other. It follows that a model-based development approach requires that common assumptions and definitions are recognized and maintained in a consistent fashion. We have investigated ways on how to automate the issue of identifying model inconsistencies and to this end we have found that automated inconsistency detection depends heavily on the ability to locate product information common to multiple models – also known as the traceability problem. This paper therefore discusses a technique where scenario executions and their observations are used to cross-reference, transform, and analyze models. This technique is useful for forward engineering and reverse engineering.

USC-CSE-2000-518 (technical report)

Extending UML for Automated Consistency Checking

Alexander Egyed and Paul Gruenbacher

The Unified Modeling Language (UML) supports a wide range of diagrammatic and textual views for modeling software development concerns. UML views are independent but connected; its meta-model enables their description under a common roof. Despite the fact that the standard seems to converge, the past UML conferences showed that researchers and practitioners alike have even grander plans for the future of UML. In this paper, we will discuss the problem of consistency checking within UML and show how the current UML standard can support it. Our finding is that UML currently exhibits a series of deficiencies with a particularly negative impact onto scalability. Resolving those deficiencies requires the adaptation and the augmentation of UML.

USC-CSE-2000-519 (technical report)

Validating Consistency between Architecture and Design Descriptions

Alexander Egyed

No abstract.

PhD Dissertation

Detecting Architectural Mismatches During Systems Composition

Cristina Gacek

The USC Architect's Automated Assistant (AAA) tool and method version 0.1 provides a capability for early detection of software architectural style mismatches among four architectural styles: Main-Subroutine, Pipe-and-Filter, Event-Based, and Distributed Processes. For these four styles, mismatch detection is based on a set of seven conceptual features distinguishing each style, and a set of bridging connectors characterizing compositions among the four styles. However, it was a significant open question whether these conceptual features and connectors were sufficient to characterize composition of other architectural styles.

The work presented here formalizes some additional architectural styles--namely Blackboard, Closed-Loop Feedback Control, Logic Programming, Real-Time, Rule-Based, Transactional Database, and Internet Distributed Entities styles--and extends the mismatch analysis capability to cover interactions of the original four styles with the new ones. The analysis results tested various hypotheses, such as the extensibility of the conceptual feature framework for mismatch detection, and the sufficiency of the original seven conceptual features to characterize the broader set of styles and their composition.

In our work we found that the underlying conceptual feature framework could work to cover a broader range of styles and systems, with some extensions. However, the conceptual feature set and the underlying Z-language formal model were not sufficient to cover the full range of styles and systems interactions.

We have developed extensions to the conceptual feature set and Z formal model to cover the full set of compositional interactions analyzed. Additionally, we provide means for checking each and every mismatch at the model level, including the dynamic ones, as well as a fully operational tool.

We also provide an initial discussion of a more formal basis for detecting and classifying architectural conceptual features, thus providing a formal framework for extending the models.

PhD Dissertation

Conflict Identification and Resolution for Software Attribute Requirements

Hoh In

A critical success factor in requirements engineering involves determining and resolving conflicts among candidate system requirements proposed by multiple stakeholders. Many software projects have failed due to requirements conflicts among the stakeholders.

The WinWin system developed at USC provides an approach for resolving requirements conflicts among the stakeholders. The WinWin system provides a framework for negotiation between the stakeholders to identify and resolve these conflicts. However, such systems do not scale well for large software projects containing many requirements.

Based on an analysis of the options for addressing this problem, I have focused on semiautomated tools and techniques for identifying and resolving conflicts among software quality attributes. I have developed two prototype support tools, QARCC and S-COST, which expand the capabilities of the WinWin system. QARCC focuses on software architecture strategies for achieving quality attribute objectives. S-COST focuses on tradeoffs among software cost, functionality, and other quality attributes. I have also developed portions of underlying theories and models which serve as the basis for the prototype tools.

Finally, I evaluated the theories, models, and tools with the results of WinWin negotiations, such as the CS577 15-project samples.

PhD Dissertation

Heterogeneous View Integration and its Automation

Egyed Alexander

Software systems are characterized by unprecedented complexity. One effective means of dealing with that complexity is to consider a system from a particular perspective, or view (e.g., architecture or design diagram). Views enable software developers to reduce the amount of information they have to deal with at any given time. They enable this by utilizing a divide-and-conquer strategy that allows large-scale software development problems to be broken up into smaller, more comprehensible pieces. Individual development issues can then be evaluated without the need of access to the whole body of knowledge about a given software system. The major drawback of views is that development concerns cannot truly be investigated by themselves, since concerns tend to affect one another. Successful and precise product development supported via multiple views requires that common assumptions and definitions are recognized and maintained in a consistent fashion. In other words, having views with inconsistent assumptions about a system's expected environment reduces their usefulness and possibly renders invalid solutions based on them.

Developing software systems therefore requires more than what general-purpose software development models can provide today. Development is about modeling, solving, and interpreting, and in doing so a major emphasis is placed on mismatch identification and reconciliation within and among diagrammatic and textual views. Our work introduces a view integration framework and demonstrates how its activities enable view comparison in a more scalable and reliable fashion. Our framework extends the comparison activity with mapping and transformation to define the 'what' and the 'how' of view integration. We will demonstrate the

use of our framework on the Unified Modeling Language (UML), which has become a de-facto standard for object-oriented software development. In this context we will describe causes of model inconsistencies among UML views, and show how integration techniques can be applied to identify and resolve them in a more automated fashion. Our framework is tool supported.

GSAW 97, El Segundo, CA

Proceedings of the Ground System Architectures Workshop (GSAW 97)

Judy Kerner, Ed.

GSAW 98, El Segundo, CA

Proceedings of the 2nd Ground System Architectures Workshop (GSAW 98)

Judy Kerner, Ed.

GSAW 99, El Segundo, CA

Proceedings of the 3rd Ground System Architectures Workshop (GSAW 99)

Sergio Alvarado, Ed.

Technical Report, The Aerospace Corporation, ATR-99(7470)-1

Evaluation Criteria for Satellite Ground System Architectures

Charles Simmons

This report presents criteria for evaluating satellite ground system architectures and development efforts. Although much work has been done on the quantitative evaluation of architectures, this report focuses primarily on manual, qualitative, heuristic evaluation - an area that has received much less attention.

References

1. Boehm, B. and Bose, P.: "A Collaborative Spiral Process Model based on Theory W," *Proceedings of the 3rd International Conference on Software Processes*, 1994.
2. Boehm, B. and Egyed, A.: "WinWin Requirements Negotiation Processes: A Multi-Project Analysis," *Proceedings of the 5th International Conference on Software Processes (ICSP)*, pp. 125-136, June 1998.
3. Boehm, B., Port, D.: "Escaping the Software Tar Pit: Model Clashes and How to Avoid Them," *ACM Software Engineering Notes*, pp. 36-48, January 1999a.
4. Boehm, B., Port, D.: "When Models Collide: Lessons from Software Systems Analysis," *IT Professional*, pp. 49-56, January 1999b-February 1999b.
5. Boehm, B. W., Bose, P., Horowitz, E., and Lee, M. J.: "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach," *Proceedings of 17th International Conference on Software Engineering (ICSE 17)*, pp. 243-253, April 1995.
6. Egyed, A., Boehm, B.: "Comparing Software System Requirements Negotiation Patterns," *Systems Engineering Journal*, 6(1), pp. 1-14, June 1999.
7. Kellner, M.: "Tutorial: Developing and Documenting Improved Software Engineering Processes," *SEPG Conference '99*, 1999.

DISTRIBUTION LIST

addresses	number of copies
ROGER J. DZIEGIEL, JR AFRL/IFTD 525 BROOKS ROAD ROME, NY 13441-4505	10
DR. BARRY BOEHM USC, LOS ANGELES 941 W. 37TH PLACE SAL ROOM 328 LOS ANGELES, CA 90089-0781	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P-STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/HESC-TDC 2698 G STREET, BLDG 190 WRIGHT-PATTERSON AFB OH 45433-7604	1

ATTN: SMDC IM PL
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

1

COMMANDER, CODE 4TL000D
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

1

CDR, US ARMY AVIATION & MISSILE CMD
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

2

REPORT LIBRARY
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

1

ATTN: D'BORAH HART
AVIATION BRANCH SVC 122-10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591

1

AFIWC/MSY
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

1

ATTN: KAROLA M. YOURISON
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213

1

USAF/AIR FORCE RESEARCH LABORATORY
AFRL/VSSOA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSOM AFB MA 01731-3004

1

ATTN: EILEEN LADUKE/D460
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

1

OUSD(P)/DTSA/DUTD
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

1

AFRL/IFT
525 BROOKS ROAD
ROME, NY 13441-4505

1

AFRL/IFTM
525 BROOKS ROAD
ROME, NY 13441-4505

1

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.